DPhil Transfer Report

SEGMENTATION OF ABDOMINAL ORGANS AND GROWTH MODELLING OF TUMOURS IN RENAL CANCER PATIENTS

Stuart M. Golodetz

stuart.golodetz@comlab.ox.ac.uk

Oxford University Computing Laboratory

Contents

Acknowledgements

I would like to thank my DPhil supervisors, Dr. Stephen Cameron and Dr. Irina Voiculescu, for their guidance and understanding this year.

I would also like to thank Dr. Andrew Protheroe, Dr. Zoe Traill, Mr. Mark Sullivan and Mr. David Cranston of the Churchill Hospital, Oxford, for their invaluable help and support, and Professor Sir Michael Brady of the Oxford University Engineering Department for his help in putting us in touch with them.

Finally, I would like to thank Clare Chennells, Sue Royall, Rosemarie Llewellyn, Rachel Shaw, Carolyn Nelson and Louise Simpson for their help in organising our meetings at the Churchill.

Chapter 1

Introduction

1.1 Research

1.1.1 Proposed Work

Despite major and prolonged research efforts around the world, cancer remains one of the leading causes of human mortality in the 21st century. Great progress has been made in treating certain cancers (e.g. breast cancer), partly as a result of extensive screening programmes and improvements in imaging techniques allowing the tumour to be caught early before it has metastasized to other parts of the body. However, the prognosis for other types of cancer which are harder to detect early (e.g. kidney cancer) is more bleak.

There are at least three ways in which improvements to this situation can come about:

1. Medics and biochemists can develop new drugs to fight cancer (e.g. *anti-angiogenic* drugs which try and cut off the blood supply to a tumour) and surgical techniques can be improved (e.g. through robotic surgery). Things like this directly affect the outcome for patients.

2. Screening programmes can be introduced to try and detect tumours at an early stage when they are more easily treated. A comprehensive screening programme using MRI (not CT!) scanning might have a major impact, but the costs for minority cancers such as renal cancer are (lamentably, in my opinion) currently seen as prohibitive, despite the obvious savings in treatment costs which would counterbalance the cost of the scans.

3. Computer scientists and mathematicians can work with medics to analyse medical images and produce tumour models. This helps medics to make the best use of the images they have available, allowing them to visualize the state of a tumour more clearly before deciding how best to treat it.

As a computer scientist, my primary interest is in the last of these areas. My aim is to develop tools that will be clinically useful, and to that end I have been talking to some of the cancer medics at the Churchill Hospital, Oxford, with a view to ascertaining the sort of tool which would help them in practice. They have shown a serious interest in (among other things) my developing a tool to measure the amount of necrosis in a tumour, and I believe this is a feasible medium-term goal. One of the prerequisites for this work is the ability to segment kidney tumours from abdominal CT scans. I have already spent three months working on this and am starting to obtain some promising results, although more work still needs to be done in this area.

Whilst developing tools which will actually see clinical use is both important and interesting, my supervisors and I also recognise that the final product of my doctorate is a thesis in which I will need to present some serious, novel research. For that reason, we aim to incorporate the work which is directly relevant to the medics into a larger body of work which can form the basis for my doctorate. In particular, we are very interested in looking at *geometric* modelling of tumour growth.

Modelling tumour growth geometrically is significantly different from modelling it mathematically. The aim is not to try and model what is happening inside a tumour using complicated modelling at the cellular level, but to model the shape of a tumour at the macro level. We have been particularly intrigued by the work done by Professor Sir Michael Brady's group in this area. In [26], they developed an algorithm which fits an optimum set of spheroids to heterogeneous liver tumours identified on CT and MRI scans. Their suggestion was that different spheroids in their model represent different *clonal centres* of a tumour. Each such clonal centre represents a sub-population of the tumour with its own properties and (crucially) level of aggressiveness. They showed that estimating response to therapy by comparing tumour volumes before and after treatment could be misleading, and that it is crucial to look at the clonal composition of the tumour post-therapy when analysing how successful a treatment has been.

We are interested in seeing whether a similar approach can be applied to renal cancer, and believe it could form a suitable topic for my doctorate. Since fitting a model to a tumour necessitates segmenting the tumour first, this will make direct use of my existing work in this area.

1.1.2 Work Done

My work thus far has been divided into two main strands. The first was a year-long mini-project I did on long-hair dynamics whilst determining a suitable topic for my doctoral thesis. It isn't directly relevant to my current work, but a summary is included in Appendix A.

Since finishing my work on hair in the summer, I have been attempting to find a niche in medical imaging/modelling. This has involved investigating a large variety of different fields; in particular, I've looked in passing at image registration, mesh visualization, mesh decimation and finite element modelling. My main focus so far, however, has been on image segmentation. I believe this ties in well with both my work for the medics and my suggested work on geometric tumour growth modelling.

1.2 Coursework and Teaching

In the first year of my DPhil, I took a number of courses in the Computing Laboratory (*Numerical Solution of Differential Equations I* and *Numerical Computing*) and the Engineering Department (*Computer Vision*, *Medical Image Analysis* and *Biomedical Ultrasonics*). The computing courses in particular helped me build on the knowledge I derived from an unexamined course I took in *Numerical Analysis* in my third year of undergraduate studies. I also took an unexamined *Complexity* course in my third year.

With regard to teaching, I demonstrated undergraduate practicals in *Computer Graphics* (both last year and this year), *Concurrency* and *Functional Programming* (last year only). Last term I ran the practicals for the new *Computer Animation* course. In Trinity 2007, I gave a revision tutorial in *Computer Graphics* to a third-year undergraduate, and last term I tutored six students in *Design and Analysis of Algorithms*.

1.3 Presentations and Lecturing

I gave three talks on hair dynamics in my first year, two to the Spatial Reasoning Group and one to the Computational Biology Group, as well as presenting several interesting papers from the research literature at group paper reading sessions. In Michaelmas 2007, I gave an *Intelligent Systems I* lecture to a third-year audience.

1.4 Other Publications

In addition to my other work, I have written a mini-series [17] on the links between template metaprogramming in C++ and functional programming for *Overload* magazine, a journal of the Association of C and C++ Users (ACCU). I have also written a mini-series [18] on the watershed and waterfall image processing algorithms for the same journal.

Chapter 2

Literature Review

2.1 Segmentation

2.1.1 Introduction

Segmentation is a sub-field of image processing in which we try to partition an image into regions which correspond to interesting, or salient, features in the image. For instance, in medical image processing, we might try and segment a CT scan into regions which correspond to axial slices through major organs, e.g. the liver or a kidney. Segmentation is rarely useful on its own; rather, it tends to occur as a pre-processing step whose output (the partition of the image) is then used as input to later stages of an algorithm. An example usage is in the 3D visualization of organs, where the segmentation results for a series of slices can be used to identify which voxels in a volumetric dataset are contained within a given organ: a mesh can then be generated from this using any suitable algorithm from the visualization literature, e.g. [11, 38, 75].

Many different approaches are used to segment images. There is no 'best' method which works well for every segmentation problem, so it is vital to select a method carefully based on the characteristics of the images under consideration. Since I will be dealing exclusively with *medical* image segmentation of abdominal scans for the purposes of my thesis, I will only discuss techniques applicable in that domain in this review, but even with that restriction in place, there are several different image modalities (e.g. CT, MRI, US) which may be encountered and a large number of different segmentation methods in use [51, 68].

Each segmentation algorithm has its own characteristics (e.g. resilience to noise, connectedness of output, etc.), making it more or less suitable for a given problem. Some algorithms output region boundaries (contours) rather than an actual partition of the image [25, 37, 76]: this may or may not be desirable in a given context (although it is worth noting that is often possible to subsequently convert

between region-based and boundary-based representations). Furthermore, the degree of automation of the algorithms varies. At one end of the spectrum, images can be manually segmented by a radiologist: this generally gives excellent results (although they may not necessarily be entirely consistent between different radiologists) but requires more human interaction, although radiologists have got this down to a fine art. At the other extreme, we can try and develop algorithms which segment images automatically [31, 36, 40, 42, 48, 62, 65, 66]: these reduce the burden on the human user, but it is rare that results are attained which are comparable with those which can be produced manually, not least because radiologists sometimes have to rely on anatomically-informed judgement to decide where the boundary of an object of interest lies and it is difficult to incorporate this expert knowledge into a computer program. That being said, some teams have achieved exceptionally good results in this area. Of particular note is the work done by Luc Soler's team in France [62] which aims to delineate anatomical structures relevant to hepatic surgery. Clinical validation on over 30 patients has shown that their fully-automatic segmentation technique on 2mm-thick enhanced spiral abdominal CT scans generates results which are often actually *better* than the manual segmentation produced by a radiologist.

In general, though, some degree of interactivity in segmentation procedures currently remains desirable, whether this merely involves allowing a radiologist to make adjustments to the output, or actually involves requesting user input to the algorithm itself (for example, a region growing algorithm might require seed points). There is a lot of overlap between the methods used for fully-automatic and semi-automatic segmentation — in general, fully-automatic segmentation tends to focus (not surprisingly) on automating the difficult-to-automate aspects of semi-automatic algorithms — so I will review approaches from both sub-fields in the following sections.

2.1.2 Thresholding

In principle, thresholding segmentation methods are quite simple. The idea is to divide the pixels of the image into classes through the use of one or more dividing lines on the image histogram. Dividing the image into two classes is known as *binary thresholding*; where more classes are used, we refer to the process as *multi-thresholding*. As an example, we could use binary thresholding to divide an 8-bit image (with grey levels ranging from 0=black to 255=white) into two classes, one containing pixels greater than (say) 160, and representing the foreground of the image, and the other containing all the

remaining pixels and representing the background.

The difficulty encountered in practice is how to determine the optimum threshold location(s). A misplaced threshold will cause an inaccurate segmentation, so choosing the location appropriately is essential. In our example above, choosing a threshold which was too high would mean that the foreground of the image would be *undersegmented* (i.e. pixels which should be classified as being part of the foreground are incorrectly classified as background) and the background would be *oversegmented* (the converse). Choosing too low a threshold would cause the opposite problem.

Owing to difficulties like this, applications are often designed so that thresholds can be chosen interactively by the user, but a great deal of work has also been done on automatically determining good threshold locations. As surveyed in [60], there are six types of approach to the problem in current use, including:

1. *Histogram shape-based methods.* These use shape properties of the histogram to find a good threshold. For instance, Rosenfeld's histogram concavity method, cited in [30], works by examining the difference between a histogram and its convex hull. A grey level at which the the height difference between the histogram and its convex hull is greatest (i.e. a point of deepest concavity) is picked as the threshold value.

2. *Clustering-based methods.* These try and group the grey level data into a given number of clusters (two in the case of binary thresholding). One example is the method of Ridler and Calvard [56]. Their idea was to take a grey-level image and produce an initial binary classification which makes the assumption that the object of interest is somewhere in the middle of the image and the corners of the image contain only background. The means of the pixels currently classified as background and object are calculated and the average of the two means is taken. The new value is then used to threshold the image and produce a new binary classification into background and object classes: it is assumed that this will be more accurate than the initial guess. Finally, the process is iterated until there is little or no change in the binary classification, and the last threshold in the iteration is chosen for use.

3. *Entropy-based methods.* These are based on information theory and pick thresholds by (for example) trying to maximise the information content in the thresholded image. As described in [73], the simplest possible method looks at two probabilities, $F(T)$ and $F^*(T) = 1 - F(T)$, each parameterised in terms of a threshold, T. The first, $F(T)$, gives us the probability of a

given pixel having a grey value less than or equal to the threshold, and the second, $F^*(T)$, gives us the probability of the value being greater than it. The information content in the thresholded image is given by

$$H(T) = -F(T)\log_2 F(T) - F^*(T)\log_2 F^*(T)$$

and attains a maximum when $F(T) = 0.5$. This is equivalent to saying that in the absence of any other knowledge, the maximum entropy principle tells us that the information contained in the thresholded image is maximised by picking a threshold which classifies half the pixels as background and half as foreground. This makes intuitive sense, but is too simplistic an approach for the majority of applications. Better alternatives have been developed, but are beyond the scope of this report.

In spite of the large amount of work done on thresholding, however, it has some significant downsides when used on its own to process medical images:

- It divides the image into two or more groups of pixels, based on their grey values, but there is no guarantee (or even an expectation) that any of these groups will be contiguous in the image. For instance, trying to segment a kidney from a CT scan by bounding it between two grey value thresholds might also result in inadvertently segmenting blood vessels across the image as well (their grey levels are quite similar to those of the kidneys). Not only are these blood vessels not part of the kidney, they are actually physically separated from it in the image! (It is also worth noting that trying to segment a kidney will generally result in segmenting both of them at once, since their grey level ranges are the same. This sort of problem can be overcome by specifying the side of the body in which we're interested.)

- It is by no means the case that acceptable threshold locations always exist. If the grey value ranges of different objects of interest significantly overlap, it may be impossible to separate them using thresholding alone.

These limitations can often be overcome by combining thresholding with other techniques. For instance, the results of thresholding often have gaps in them, which can sometimes be filled in by carefully applying various morphological operators (e.g. morphological opening and closing). Luc Soler's team [62] made use of thresholding (as one technique among many) and achieved excellent automatic segmentation results for the liver. However, they did not use thresholding on its own. For

instance, they segmented bones by thresholding for bright areas and then keeping only those which were near to the fat tissue (which had already been segmented). Simple thresholding alone would have been insufficient for the task, since structures such as the aorta also appeared bright on the contrast-enhanced images.

2.1.3 Region Growing

Region growing methods for segmentation essentially work as follows. First, an initial seed point is chosen for a feature of interest. Then, the region is 'grown' by iteratively considering all points which are adjacent to the region and adding any which satisfy certain criteria. For example, we could choose to add adjacent pixels whose grey value differs from that of their neighbour in the region by less than a certain amount. Alternatively, we could try and add adjacent pixels which preserve the homogeneity of the entire region (for some suitable definition of homogeneity). A basic region growing algorithm can be implemented straightforwardly using a queue. Starting from a queue containing only the initial seed point, we repeatedly pop the pixel at the front of the queue, consider its non-region neighbours for addition to the region, and push any which satisfy the requisite criteria onto the end of the queue. The process terminates when the queue empties.

The key issues when implementing region growing are how to choose the seed point, how to formulate the criteria specifying which points to add to the region, and how to decide when the process should terminate. For automated segmentation methods, how to choose the seed point is of fundamental importance; semi-automated algorithms can focus exclusively on the latter two problems, relying instead on the user to interactively specify an initial seed.

As mentioned above, one of the simplest approaches to region growing is to add adjacent pixels which are within a certain fixed threshold value of their neighbour in the region. Practical region growing methods, however, such as [36, 53], tend instead to use *adaptive* region growing, whereby the criterion varies to take account of the area around the pixel under consideration. In [36], for example, the approach taken is as follows. After locating an initial seed point (s_x, s_y), a 7×7 mesh is placed over it and the maximum and minimum pixel intensities within the mesh, $M(s_x, s_y)$ and $m(s_x, s_y)$ are determined. From these, a contrast range $t_0 = M(s_x, s_y) - m(s_x, s_y)$ is calculated and recorded. Next, for each pixel (x, y) under consideration for addition to the region, values $M(x, y)$ and $m(x, y)$ are similarly calculated, and a local value $\theta_{\text{local}} = (M(x, y) + m(x, y))/2$ is determined.

The region growing criterion is then formulated as $|f(x,y) - \theta_{\text{local}}| \leq t_0$, i.e. we add an adjacent pixel (x,y) to the region if the absolute difference between its grey value $f(x,y)$ and the midpoint of the contrast range of the 7×7 mesh surrounding it is less than the contrast range of the 7×7 mesh centred on the initial seed point. The region growing is specified to stop when this absolute difference is greater than a certain threshold, implying that the area surrounding a given pixel is not homogeneous.

The advantages of region growing methods are that the resultant region is guaranteed to be connected in the image (unlike with thresholding) and that they are, on the whole, fairly easy to implement. However, from the point of view of automatic segmentation, they present difficulties, because choosing an initial seed point is in general a non-trivial problem. The usual approach taken for automatic seed point selection is to rely on statistical data about where the features of interest (e.g. organs) usually lie in the body. For instance, the approach in [36] is to search for suitable seed points in two elliptical regions on each side of the body, one for each kidney. This works quite well, but doesn't seem as if it would be that robust if tested on unusual cases.

Whilst region growing algorithms are guaranteed to produce a connected result, the region may still have holes in the middle of it. Whilst this may be desirable if we really are trying to segment a torus-shaped feature, on the whole we need to post-process the region growing results to remove these holes. Common techniques for doing this include morphological closing, etc.

2.1.4 Approaches from Mathematical Morphology

In the words of some of its practitioners[1], the field of mathematical morphology espouses a 'non-linear approach to image processing'. Its interesting techniques include dilation, erosion, and morphological opening and closing, but our emphasis in this report will be on one particularly useful morphological technique for image segmentation: the watershed transform.

The Watershed Transform: Techniques

Note: I have written an article on the watershed transform for ACCU's Overload journal, which appears as Appendix B in this report. For that reason, I plan to give only a limited presentation of the technique here. Readers are referred to the appendix for more information.

[1] The Centre for Mathematical Morphology at the Ecole des Mines de Paris.

The idea of the watershed transform is to view an image as a landscape and divide it up into valleys. Each valley in the landscape will correspond to a region in the segmentation result. We don't perform the transform on the actual image to be segmented, since its valleys may not correspond to the features we're interested in. Instead, we take the initial image and try and generate a derived image from it in which the valleys correspond to our features of interest: in the case of medical images, this often involves something similar to taking the gradient of the initial image, since organs tend to look fairly homogeneous on scans (and thus the magnitude of the gradient within them is low). The derived image then becomes the input to the watershed transform itself.

In terms of implementation, the watershed transform can be viewed in one of two different ways:

1. *A flooding/immersion process*. Imagine poking holes in the local minima of the landscape and then lowering the landscape vertically into a lake. As the landscape is lowered, water will seep through the holes and catchment basins will form in each of the valleys of the image. As the water continues to rise, some of the catchment basins will meet: at this point, we add dams, or watersheds, to keep them apart, and continue lowering the landscape. At the end of the process, the dams we added will act as dividing lines between the valleys, thus segmenting the underlying image into regions.

2. *A rainfall process*. Imagine dropping a raindrop at each point in the landscape. Assuming the image has no (non-minimal) plateaux (flat regions), the drop will run downhill, via a path of steepest descent, until it ends up in a local minimum. (There are ways of dealing with non-minimal plateaux, e.g. the method described in the appendix.) The point at which it started will be associated with this local minimum (it will be called part of the minimum's catchment basin) and the process will continue until all the points in the landscape have been processed. The result will be a region-based partition of the landscape (and hence the underlying image) into catchment basins, one per local minimum.

These two alternative ways of viewing the process have led to two different classes of techniques. In particular, [6] and [54] are examples of the former approach, and examples of the latter can be found in [42, 47, 63]. It is worth noting that the various definitions are not always exactly equivalent. For example, some approaches generate results containing watershed lines, whilst others generate only regions. Also, rainfalling approaches don't generally give identical results to their immersion-based counterparts. It is thus important to choose an appropriate method with care.

The Watershed Transform: Pre-Processing

The biggest problem with the watershed transform as a segmentation approach is that it is overly affected by spurious local minima in the landscape. This leads to massive over-segmentation of the image. To explain the problem, consider an analogy. If we try and divide a dimpled landscape into valleys, counting each dimple (small hollow) in the landscape as a valley, then we will end up with vast numbers of 'irrelevant' valleys at the end of the process, obscuring the more significant valleys we are trying to find: this is essentially the problem faced by the watershed transform. The 'dimples' in an image can be caused by things like noise, but even in the absence of image artifacts, there will usually be valleys corresponding to features in which we have no interest. We need a way of separating the wheat from the chaff.

To overcome this difficulty, various approaches have been suggested in the literature. The effects of less prominent noise can be mitigated to an extent by smoothing the image, although care must be taken not to blur the edges in the image when doing this. Another approach [3] is to suppress some of the irrelevant contours in the gradient image by determining the connected components of the gradient image and removing (by setting the pixels of the component to zero) all those whose size is less than a threshold (e.g. the perimeter of the smallest object of interest).

One technique which works quite well in practice is to apply spatially-variant smoothing to the image: instead of smoothing uniformly over the entire image, the idea is to smooth less in areas where we think there may be edges. This is the idea behind non-linear diffusion filters [50]. (In Chapter 3, I describe another approach based on similar principles.)

The Watershed Transform: Post-Processing

The approaches described so far try to pre-process an image (or its gradient image) to reduce watershed over-segmentation. Post-processing the watershed results is also possible, and in general a combination of both techniques tends to be employed.

The general idea of watershed post-processing is to try and merge the large number of generated regions together in a way that generates larger regions corresponding to our objects of interest. Attempts can be made to group together similar regions using fuzzy relations, e.g. [49].

The idea of region-merging can also be used to create partition hierarchies, intuitively corresponding to a series of segmentations of an image at different scales. For example, the waterfall

algorithm [5, 40] merges regions by iteratively running a watershed-from-markers algorithm on the region adjacency graph of the watershed result. (Note that I have described it in much more detail in Appendix C.) This is particularly useful when trying to pick out organs of different sizes. A similar approach is described in [72].

The type of hierarchy generated by the waterfall algorithm is by no means the only possible hierarchy, and indeed it may converge (to a single region) too fast for some applications. Alternative hierarchies are also possible, e.g. ones based on the saliency of contours in the watershed results [46].[2]

2.1.5 Deformable Models

As their name suggests, deformable models methods work by taking an initial model of the features under consideration and deforming it to fit the actual data available. The initialisation of the model is generally based on *a priori* anatomical knowledge. The model itself can be represented in a variety of ways, and these ways correspond to a number of different approaches to the problem. For example, the snakes method, which we will see shortly, represents the model as a parametrically-defined spline, and is hence referred to as an example of a parametric deformable model (it is sometimes also referred to as an explicit deformable model). In contrast, level sets represent the model *implicitly* and are an example of implicit deformable models. Implicit deformable models were developed more recently than their parametric counterparts, but both types see a lot of use.

Snakes

As originally defined by Kass, Witkin and Terzopoulos in [25], "A snake is an energy-minimizing spline guided by external constraint forces and influenced by image forces that pull it toward features such as lines and edges."

Essentially, the idea works as follows. We represent the position of the snake, in terms of a spline parameter s, as $\mathbf{v}(s) = (x(s), y(s))$, and define an *energy functional*, E^*_{snake}, as

$$E^*_{snake} = \int_0^1 E_{int}(\mathbf{v}(s)) + E_{image}(\mathbf{v}(s)) + E_{con}(\mathbf{v}(s))\ ds.$$

[2]Whilst this is generally a good paper, it is worth noting that it contained an error which had to be corrected later in a comment [35]. An improved correction was then published by one of the original authors [57]. All three should be read together to avoid confusion.

The three terms in the summation are *internal* energy, which tries to limit the curvature of the snake, *image* energy, which tries to attract the snake towards features in the image such as lines and edges, and *constraint* energy, which allows the user to apply constraints to influence the result of the segmentation. The snake algorithm as a whole attempts to find a spline minimising E^*_{snake}.

The original snakes paper describes a continuous model, but more recent research [37, 44, 45] has seen the development of discrete models as well. The model described by Lobregt and Viergever in [37], called a *discrete dynamic contour model*, is particularly interesting, both because it is intuitively easy to understand, and because it links in with my previous work. It is thus worth describing in more detail.

Rather than relying on ideas of energy-minimization, Lobregt and Viergever model snakes using a force-based physical simulation (note that this ties in well with my earlier work on physical simulations of long hair [19]). In their approach, a snake is a set of vertices connected by edges. At each time-step, various forces are applied to each vertex, which gradually *deform* the snake towards the desired result. The results of this approach will depend to an extent on the lengths of the edges joining the vertices: if an edge is too long, important image features may pass through the gaps between vertices; if it is too short, the snake may become overly fixated on small details, not to mention the speed of the process being adversely affected. For this reason, after each deformation step, the snake is *resampled* (by adding or removing vertices where necessary) to keep the edge lengths within certain limits.

The forces applied to each vertex closely mimic the energy terms in the original snakes paper. The force $\mathbf{f_i}$ applied to vertex i is defined as the weighted sum

$$\mathbf{f_i} = w_{ex}\mathbf{f_{ex,r_i}} + w_{in}\mathbf{f_{in,i}} + \underbrace{w_{damp}}_{<0}\mathbf{v_i},$$

where $\mathbf{f_{ex,r_i}}$ is an *external* force term corresponding to the image and constraint terms from the original formulation, $\mathbf{f_{in,i}}$ is an *internal* force term corresponding to the original internal term, and the remaining term is a new addition used to apply damping to try and bring the simulation to rest. (The real numbers w_{ex}, w_{in} and w_{damp} specify the weights to be given to each of these three factors. The paper tended to set all three of these to 0.5: apparently this was derived empirically.)

It is important to mention that the method relies on quite a close initialisation (i.e. image features have quite a short capture range). In [55], this problem is circumvented by peforming a watershed-

from-markers segmentation and using the result of that to initialise the snake. Another alternative, referred to there, is to try and add additional external forces to solve the problem: in particular, some success has been had with *balloon forces* [10] and *gradient vector flow* [76].

Problems of this kind with snakes methods have led to a great deal of interest in level sets as an alternative, although snakes also remain popular as a well-established and far simpler alternative. They also have the advantage that they can represent open structures as well as closed ones.

Level Sets

Level sets are to snakes what implicit representations of functions are to parametric ones. As an initial introduction, consider the alternative representations of a circle of radius r, centred at the origin. An implicit representation could be

$$x^2 + y^2 = r^2,$$

whereas a parametric representation in terms of an angle θ could be

$$\begin{aligned} x &= \cos\theta \\ y &= \sin\theta. \end{aligned}$$

Now, consider a more general form of the above. Instead of thinking about circles, consider the function $\phi(x, y) = x^2 + y^2$, which defines a scalar field over $\mathbb{R}^2$. For any value $k > 0$, the equation $\phi(x, y) = k$ defines a circle in the x-y plane: we will refer to each of these circles as an *isosurface* of ϕ, because each of them is the surface (curve) of all the points at which ϕ takes a particular value[3].

Note what happens if we change ϕ. If we redefine it as $\phi(x, y) = x^3 + y^3$, the isosurfaces change from being circles to hypercircles. This is a key idea: by changing the function ϕ, we can move the isosurfaces of ϕ around, *without having to represent them explicitly*. This is the insight behind level sets: we represent the contour we're interested in as an isosurface of some function ϕ, then modify ϕ to modify the contour. Instead of using simple functions like $x^2 + y^2$ for our ϕ, we can think of a general function $\phi : U \to \mathbb{R}$, where $U \subset \mathbb{R}^2$.

In practice, to represent contour changes over time, we make ϕ a function of time as well, giving us $\phi(\mathbf{x}, t) : U \times \mathbb{R}^+ \to \mathbb{R}$. We then set this equal to some constant k to actually specify which

[3] *Iso* means 'equal' in Greek, so here we are talking about the surface containing all the points with the same ϕ value.

isosurface of ϕ we're interested in.

Consider a simple example, that of a circle, centred at the origin, which gradually expands outwards as the time increases (see Figure 2.1).

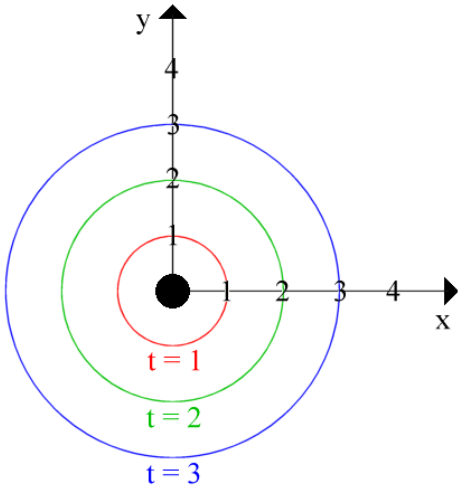


Figure 2.1: A simple circular contour which gradually expands over time

In particular, suppose its radius is t at time t. We represent this as:

$$\phi((x, y), t) = \frac{x^2 + y^2}{t^2} = 1 = k$$

For a more complicated example, consider gradually changing the circle into an ellipse over time (see Figure 2.2).

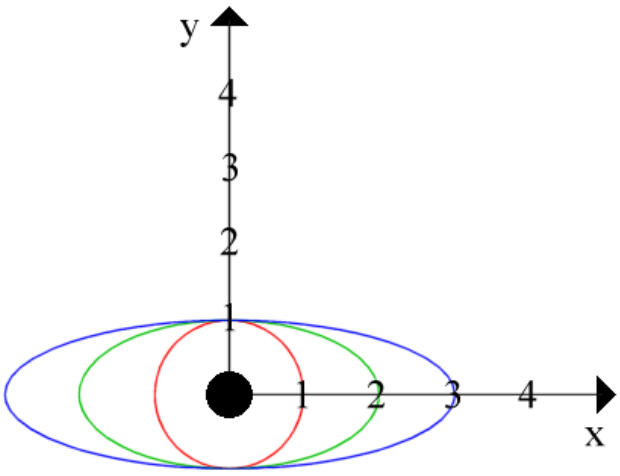


Figure 2.2: An example of a contour which changes its shape over time

This could be achieved by writing the following:

$$\phi((x, y), t) = \left(\frac{x}{t}\right)^2 + y^2 = 1 = k$$

In practice, we are almost never able to give an explicit function for ϕ: the simple examples above notwithstanding, it's just not possible for real-world applications. Instead, we discretise the process by specifying initial values for ϕ at points on a discrete grid, then derive a partial differential equation for ϕ and solve it numerically to modify the contour over time. The PDE in question can be found by taking the total derivative of

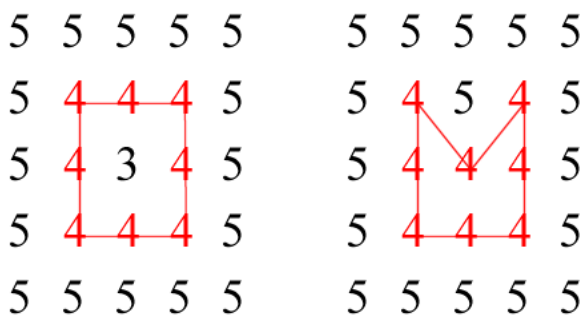
$$\phi(\mathbf{x}, t) = k,$$

giving us

$$\frac{\partial \phi}{\partial t} = -\nabla \phi \cdot \mathbf{v},$$

where $\mathbf{v} = \partial \mathbf{x}/\partial t$. By making $\mathbf{v}$ a function of the position of $\mathbf{x}$ and the geometry of the surface (which it turns out can be represented by the differential structure of ϕ), we can control how we want the contour to evolve over time.

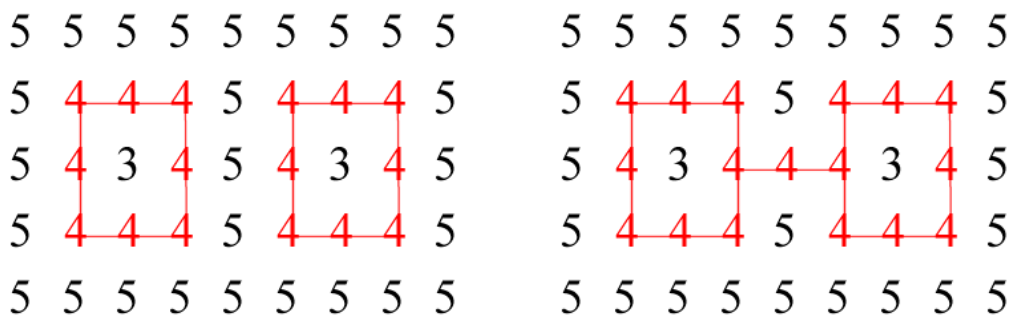
How to numerically solve the above equation is beyond the scope of this report, but before moving on, I would like to look at a concrete example of the method on a discrete grid, to illustrate some of its advantages. Consider the discrete grid of ϕ values in Figure 2.3(a). The marked points form an isosurface of ϕ, specifically the $k = 4$ isosurface. This is entirely analogous to the examples above, but on a discrete grid instead of in a continous space. By changing the grid values, as in Figure 2.3(b), we can move the isosurface however we wish.



(a) (b)

Figure 2.3: An example of level sets on a discrete grid

Now, consider the further example shown in Figure 2.4(a). Here the situation seems more complicated, in that the points on the $k = 4$ isosurface are not all connected to each other. This is actually an advantage of the level set method, however. By representing the surface implicitly, we can model surfaces with multiple connected components without having to do any additional work. Indeed, the number of components can evolve over time: in Figure 2.4(b), we see that the two components can be joined by simply changing the value of ϕ at one of the grid points to 5.



(a) (b)

Figure 2.4: An example with more than one connected component

One final advantage of level sets worth mentioning is that the method works equally well in 3D, although the extra processing involved means that clever numerical techniques are required. The details of how these work are beyond the scope of this report but, as an example, one method (the narrow-band technique) basically restricts numerical calculations to a narrow region around the isosurface we're particularly interested in. The insight is that solving the PDE over the whole domain is only necessary if we're interested in solving for the entire family of isosurfaces associated with ϕ: often we're only looking at a single isosurface, so the extra computations aren't necessary. For more details, take a look at [78].

Other Deformable Models

Other notable parametric deformable models in existence include the contribution of Tsagaan et al. [66], who used a NURBS-based model of a kidney and deformed it by minimising an energy function. Their results seem good in some cases, but differ markedly from the manually segmented results in others.

Parametric and implicit models are not the only types of deformable model in use, however. In

particular, [23] uses a deformable model based on charged particles to achieve some interesting automatic segmentation results. Here, the model is represented as a set of charged particles under the influence of an electrostatic field. The authors' work in this area is still ongoing. Yet another deformable model can be found in [52], in which the authors' make use of a medial representation known as m-reps.

2.1.6 Learning Methods

Learning segmentation methods generally start with a training phase, in which a large number of scans are taken and used to construct some variety of model, which encodes information that can be used to segment subsequent scans. Two different types of learning method will be discussed in the following subsections.

Atlas-Based Approaches

Atlas-based methods start by constructing an atlas, or reference segmentation, from a set of training data. For instance, in [48], a probabilistic atlas was constructed by registering (aligning) the manually segmented volumes of 31 patients onto that of a carefully chosen reference patient (thus the data came from 32 patients in all). The atlas was represented as a 3D grid of vector values, where the components of each vector corresponded to the organs under consideration, and the values of the components indicated the fractional percentage of registered data sets in which the point was labelled as the given organ. As an example, if we were considering the liver and the two kidneys, the vector $(0.4, 0.6, 0)^T$ at a point could indicate that the point was inside the liver in 40% of the data sets and inside the right kidney in 60% of them.

After constructing such an atlas, new sets of data can be segmented by registering them onto it. In [48], they use a *maximum a posteriori* (MAP) approach after registration to find the segmentation result which best explains the observed data. The important point here is that the segmentation result is based on both the data for the individual patient and the information encoded in the atlas. The atlas provides the prior probabilities at each voxel, and these are refined in light of the actual patient data.

To use the notation in [48], we denote the segmentation result field as $\mathbf{X}$, the field of observed data for the individual patient as $\mathbf{Y}$, and the probabilistic atlas as $\mathbf{A}$. Each of these represents a volume of N voxels, indexed linearly (in some order) from 1 to N. So for instance, a particular segmentation

result could be given by $\mathbf{x} = (x_1, x_2, \ldots, x_N)$. Our aim is to find the best possible $\mathbf{x}$, defined as

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmax}}\, \mathbf{P}(\mathbf{X} = \mathbf{x} | \mathbf{Y} = \mathbf{y}).$$

In other words, we seek the segmentation result which best explains the observed patient data, $\mathbf{y}$. The probabilistic atlas is used in all of this as the prior for $\mathbf{X}$. Suppose there are L different possible voxel labels, numbered from 1 to L. Each voxel a_i of the probabilistic atlas contains an L-vector, $(a_{i,1}, \ldots, a_{i,L})$, where $a_{i,\ell}$ gives the probability of a voxel's correct label being ℓ. We use this to define the prior probabilities for $\mathbf{X}$, writing

$$P(x_i = \ell) = a_{i,\ell}.$$

In other words, the probability of the correct segmentation result for a given voxel being ℓ is given by the ℓ^{th} component of the probabilistic atlas at that location. With this link made, the result determined will depend on both the atlas and the observed data.

Neural Network Methods

Neural networks (NNs) can be used to segment images in a number of ways. A simple NN technique can be found in [67], where the authors use a feed-forward network to classify each pixel into one of three classes (liver, boundary or non-liver) according to the grey-level histogram of the 7x7 region centred on the pixel. (In practice, they use a histogram with 16 grey levels instead of 256, to avoid a proliferation of input nodes in the NN.) The schematic of how this works (borrowed from their paper) is shown in Figure 2.5.

The NN is initially trained by picking a suitable image from the middle of the volume and marking a significant number of training regions for each class. The well-known back-propagation algorithm for NNs is used to update the weights on the network arcs accordingly.

The results of the scheme are somewhat hard to evaluate since the images in this somewhat old paper (1994) have not really survived the ravages of time. However, they do seem to show that the authors have obtained something that looks reasonably like a liver, so the method has some merit.

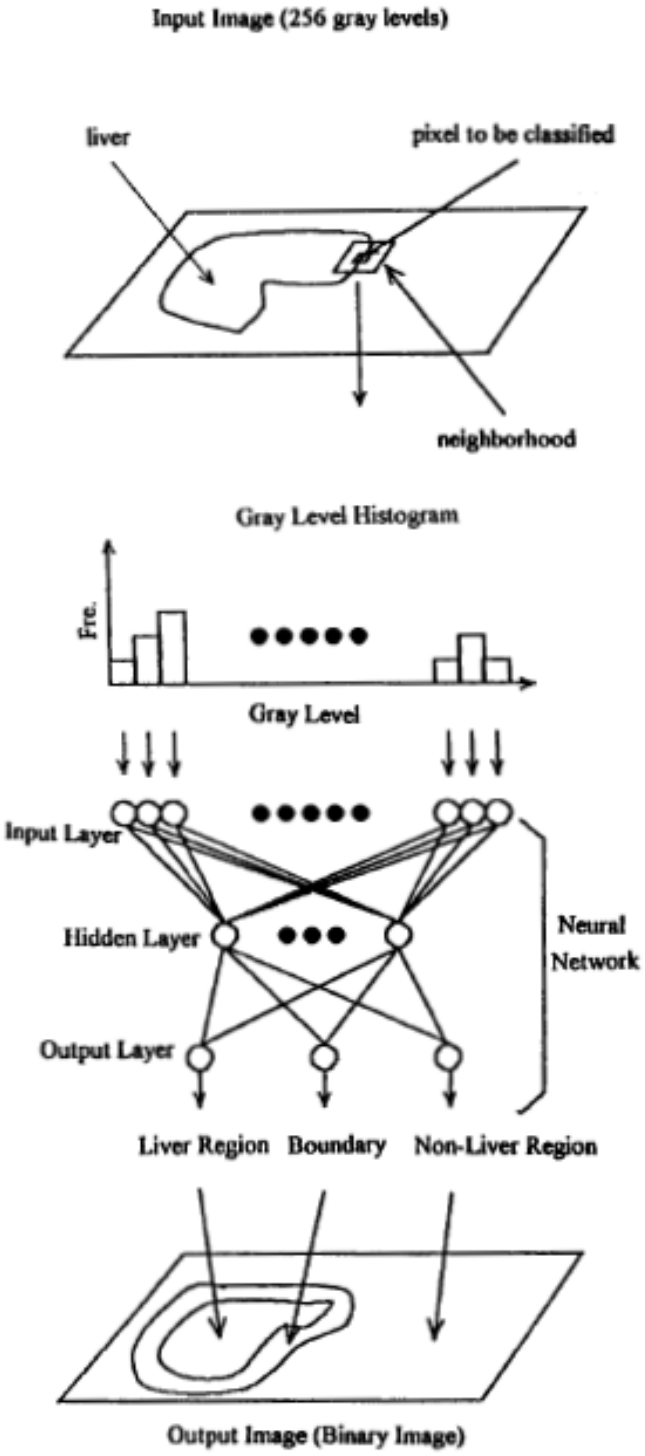


Figure 2.5: Schematic diagram of the neural-network-based boundary detection method used in [67] (borrowed from the paper in question)

A more up-to-date (and more complicated) NN scheme can be found in [31]. Here, the authors use a multi-module, recurrent neural network to segment multiple abdominal organs (see Figure 2.6). There is a module associated with each label under consideration. Each node of a given module k corresponds to a pixel in the image and encodes the probability that the pixel should be assigned label k. The weights of the network arcs are initially derived from (for example) a correlation matrix containing the likelihoods of various labels occurring next to each other. (For example, the liver might be quite likely to occur next to the right kidney, but definitely shouldn't occur next to the left kidney.) An iterative state evolution algorithm is used to determine the probabilities at each node in each of the modules of the NN. The initial probabilities are generated using something called the 'Kohonen self-organizing algorithm' (see the paper for details) and the nodes are updated at each time step based on the current probability at a given node and the support it receives from its neighbours. (So, for instance, if a pixel was currently classified as part of the left kidney, but all its neighbours were liver, it would be very likely to change to liver over time.) The results of this method (after combining it with fuzzy spatial rules for organ identification) are quite good (though the authors admit that more work is needed).

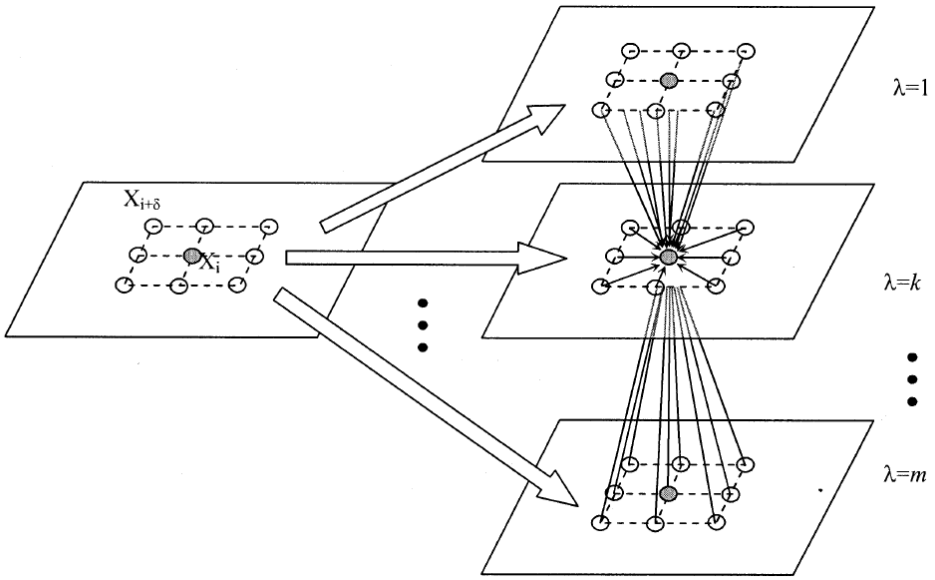


Figure 2.6: The architecture of the contextual neural network used in [31] (borrowed from the paper in question)

Figure 2.7: An automatic 3D reconstruction of patients from medical images, as performed by the IRCAD surgical planning tools

2.1.7 Hybrid Methods

It is often possible to achieve better results by combining a number of different segmentation methods than it is by using a single method on its own. A striking example of how successful this sort of technique can be is the work of Luc Soler's team in Strasbourg [62] (see Figure 2.7 for a screenshot from their surgical tool).

The cited paper represents only a small part of their work on a surgical tool, and focuses on automatically segmenting the liver. Their segmentation method first uses thresholding, morphological operators and distance maps to identify the skin, bones, lungs, kidneys and spleen. The second stage is then effectively a clever deformable model approach in which they embed a reference model of the liver into the image and automatically deform it to the liver contours. Later stages use Gaussian fitting on the image histogram to separate normal liver tissue from blood vessels and lesions (tumours), a result which is further refined by their own analytical methods. The final stage of their work makes use of higher-level medical knowledge to label the hepatic portal vein and segment the liver anatomically: these features are not visible from the scans alone. As the screenshot shows, the results of this extensively hybrid technique are excellent.

Another couple of hybrid segmentation methods can be found in [22], in which the authors combine fuzzy connectedness, Voronoi diagram classification and deformable models into one segmentation method, and Gibbs priors and deformable models into another. The details of these techniques are beyond the scope of this report, but an implementation of the methods can be found in the popular (and freely-available) *Insight Toolkit*.

2.1.8 Evaluation of Segmentation Results

In many ways, evaluating the results of a segmentation process is just as important as segmenting the image in the first place. As described in [79], there are essentially three different types of evaluation method in common use: analytical methods, empirical goodness methods and empirical discrepancy methods.

Analytical methods deal with the actual algorithms rather than their results: they focus on things like the complexity of the algorithm, or its requirements. They usually produce qualitative results that are not directly tied to the segmentation accuracy for particular applications, which is often what we really want to know.

Empirical goodness methods evaluate the segmentation results in terms of subjectively-defined 'goodness' measures. For instance, some researchers have proposed methods which rate the results on their levels of intra-region uniformity (with high uniformity more desirable) and inter-region contrast (with high contrast more desirable). The advantage of using a method like this is that it can be applied directly to the segmentation results, without reference to any external source of information like a 'gold standard' segmentation result. The disadvantage is that methods like this provide only subjective measures of segmentation quality.

In contrast, *empirical discrepancy* methods compare the segmentation results to a 'gold standard' result. This has the disadvantage of requiring us to have such a reference result (in the case of medical images, we may have to ask the radiologist to manually segment an image for us), but gives us a reasonably objective, quantitative evaluation of our algorithm in the context of a given application. One simple method in this category is to calculate the percentage of pixels misclassified. More complicated methods are also possible.

2.2 Region Classification / Organ Identification

Region classification is the task of assigning *meaning* to the regions identified by the segmentation process. As a clarifactory example, when we generate a kidney-shaped region, we're just doing segmentation, but when we identify it as a kidney we're doing region classification. The obvious way of classifying regions is to generate a series of useful properties for each region, and then use those to determine what the region might be. Useful properties might include location, area, mean grey value, elongatedness, etc. For instance, a feature which was located in the middle of the patient's back, appeared bright white on a CT scan and had an area of (say) 1000 pixels or so might well be part of the spine.

In general, designing so-called *positive* shape constraints like the above is difficult, as noted in [28], since there are very few shape invariants to rely upon. In other words, regions which should be classified as a given organ can come in a variety of different shapes. In contrast, *negative* shape constraints, which specify shapes which a region *cannot* take if it is to be classified as a given organ, can often be much more reliable and successful in this regard.

One paper which does deal with organ identification using positive shape constraints is [31], in which they use spatial fuzzy rules for organ recognition (the fuzziness helps them avoid some of the problems they would otherwise face by using positive shape constraints). Their rules were devised in collaboration with a radiologist. The results seem promising, but their method is not successful in all cases, as they admit themselves. Another relevant (if somewhat old) paper on this is [12], in which they label CT scans of the head using rule-based labelling. However, it is hard to judge the quality of their results as the images are old and rather unclear.

Aside from methods involving shape constraints, it is also possible to classify regions based on their relative spatial relationships in the image [4]. This is an interesting approach, because the spatial relationships between organs tend to exhibit much less variability than other properties such as the shapes of the organs in the images.

2.3 Tumour Necrosis

Tumour necrosis, the death of cells within a tumour, remains an imprecisely-defined phenomenon. Lee et al. [32] note that 'Controversy [...] still exists on the actual method of defining tumor necrosis',

particularly with regard to the differences between what can be seen on radiological scans, and what can be seen during pathological analysis. Nevertheless, it has long been hypothesised that tumour necrosis (howsoever it is defined) might be a useful prognostic indicator for certain types of cancer, e.g. renal cell carcinoma (RCC), the most prevalent type of kidney cancer. In other words, it is thought that the presence or absence of necrosis in a tumour might correlate (possibly in conjunction with other factors) with the length of time for which, and indeed whether, a patient survives. The reason that this might be the case remains unclear, although [32] comments that 'Some have suggested that necrosis within the tumor may be the result of the tumor rapidly proliferating and outgrowing the blood supply'. If this were true, it would certainly make sense for there to be a link between necrosis and patient survival, since the presence of necrosis would indicate a high rate of tumour growth (i.e. an aggressive tumour which would be much more likely to kill you).

A number of statistical studies (e.g. [8, 15, 13, 32, 34, 59]) have been undertaken to determine whether tumour necrosis is a useful prognostic indicator for RCC. Some other studies (e.g. [14, 43, 58]) have also included tumour necrosis in a wider investigation of which variables are most significant for patient outcome. Finally, studies have been done on the prognostic implications of tumour necrosis for other types of urological tumour (e.g. [29, 33]).

The conclusions of these studies have been inconsistent, and require careful evaluation. The view of Fetter [14] was that the presence of necrosis was a favourable tumour characteristic, but other studies have suggested either that necrosis has an adverse effect on prognosis (e.g. [13, 32, 33, 59]) or that it is not independently prognostically significant (e.g. [15, 34, 43]).

There are many potential reasons for these conflicting results. For instance, Lam et al. [13] demonstrated that in their study, tumour necrosis was an independent predictor of survival for non-metastatic RCC, but the same was not true for patients with metastatic RCC. This would seem to make some sense, since the presence of metastases is associated with low survival rates regardless of the condition of the primary tumour, necrotic or not. It has also been suggested (by Sengupta et al. [59]) that the prognostic implications of tumour necrosis may vary depending on which histological subtype of RCC is under consideration. Some studies, e.g. [32], suggest that the *extent*, rather than merely the presence, of tumour necrosis is important for prognosis, whereas others, like [13], have found the opposite. This latter result seems somewhat illogical [2].

In conclusion, the prognostic implications of tumour necrosis remain poorly understood. Tumour necrosis seems to be significant for certain non-metastatic tumours, but whilst the pathogenesis of

necrosis and its role in tumour biology remains a mystery, producing a definitive answer to the question of how it affects patient survival in general will be difficult.

2.4 Geometric Growth Modelling

When people talk about modelling tumour growth, they are usually referring to the design and use of (generally complex) mathematical models of tumour growth, e.g. [1]. This sort of modelling is often done at the cellular level, and is well beyond the scope of this report. Geometric growth modelling, by contrast, involves modelling tumour growth at a much higher level, and to the best of my knowledge is a new approach introduced by Sir Mike Brady's group in [26]. That paper is primarily focused on liver tumours, and we are interested in seeing whether the approach can be adapted to work for kidney tumours as well.

Tumours are naturally heterogeneous, i.e. they consist of multiple subpopulations / *clonal centres* of cells, each of which have different characteristics, be that in terms of individual genes, DNA ploidy, histological morphology or gross morphology. This is due to mutations during tumour development, often occurring at any early stage of tumour growth. The idea in [26] is that since, impeding factors aside, each subpopulation is likely to grow outwards in a spherical manner, the shape of a liver tumour over time can be modelled as a set of expanding spheres, each corresponding to an individual clonal centre (see Figure 2.8).

By analysing scans of the tumour at different points in time, therefore, it is possible to fit an 'optimal' set of spheres (or, more generally, spheroids) to the image of the tumour, and thereby gain an understanding of how the different subpopulations in the tumour are arranged, and their sizes. This information directly informs patient management decisions, since it can indicate, for example, that a particularly aggressive subpopulation has regressed and that the tumour is therefore responding well to therapy. This could well be missed when using existing methods for analysing a patient's response to therapy, since these tend to look only at the change in volume as a result of therapeutic intervention.

This recent (2007) paper is, to the best of the authors' knowledge, the first paper which combines image analysis, histological/genetical tumour heterogeneity and mathematical modelling in a single model. (A literature review of this area is thus, of necessity, a short one!) However, the approach is very interesting and something we would be interested in investigating further as part of my doctorate.

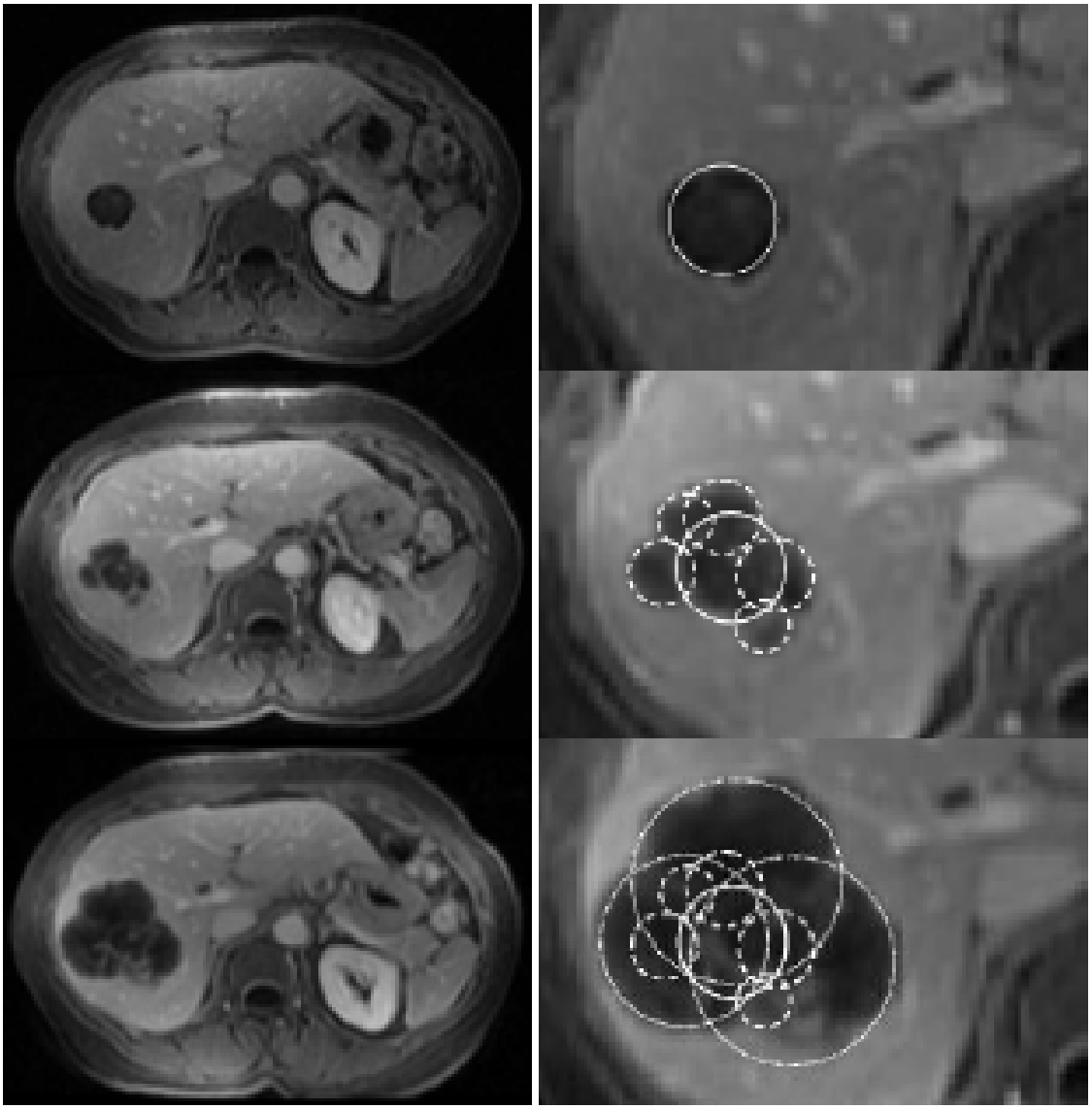


Figure 2.8: Heterogeneous tumour growth for a patient over a one year period (this appears as Figure 2 in [26] and is used here purely for explanatory purposes)

Chapter 3

Existing Work

3.1 Overview

My work to date has involved trying to segment CT volumes (made up of 5mm slices) from three different patients[1]. The current outline of my proposed algorithm is as follows:

$$\begin{array}{rl} \xrightarrow{\text{Image}} & \textsc{Hierarchy Generation} \\ \xrightarrow{\text{Partition Tree}} & \textsc{Initial Property Generation} \\ \xrightarrow{\text{Annotated Tree}} & \textsc{Feature Identification} \times n \\ \xrightarrow{\text{Features}} & \textsc{Feature Refinement} \times n \\ \xrightarrow{\text{Refined Features}} & <\textsc{Output}> \end{array}$$

The first phase of the algorithm takes an image (e.g. a slice from a CT volume) and uses it to generate a partition hierarchy. This is a series of K partitions, $P_1, \ldots, P_K$, of the image, such that each partition is formed by merging some of the regions in its predecessor (the first partition is derived from the image itself). More formally, we can represent each partition P_k as a set of mutually-disjoint regions $\{R_{k,1}, \ldots, R_{k,r(k)}\}$, where $r(k)$ denotes the number of regions in partition k. Each region in partition P_{k+1} is then either a single region from P_k, or the union of two or more such regions. We note in passing that for $1 \leq k \leq K$,

$$\bigcup_i R_{k,i} = I,$$

where I represents the entire image, and that for $1 \leq k < K$, $r(k+1) < r(k)$. The lowest partition in the hierarchy, P_1, is a very fine partition of the image; successive partitions get gradually coarser

[1] With grateful thanks to the medics at the Churchill Hospital, Oxford.

(contain fewer regions) as we near the top of the hierarchy.

As will be explained more fully in the sections which follow, we can represent a partition hierarchy as a tree structure (a *partition tree*). This is a tree in which each tree level represents a partition of the image: each tree node corresponds to a region in the partition represented by its level. The root node of the tree corresponds to a single-region partition of the image, or in other words the entire image. The second phase of the algorithm takes such a tree as its input and annotates each of its nodes with useful properties of the region it represents. For instance, we might annotate each tree node with the area of its corresponding region, or perhaps its average grey level in the image.

The output of the property generator is an annotated tree, which we can then use for feature identification. This third phase of the process tries to iteratively identify regions which represent interesting features in the image. For instance, we might identify a region in the tree which corresponds to the left kidney. We might also be interested identifying regions which are definitely not relevant to the task at hand. After a region is identified, its corresponding node will need to be removed from the tree, and the regions above it on the path to the root of the tree will need to be updated: this process will be described more fully later on.

The final stage of the algorithm takes a list of identified features and refines them using different segmentation methods, e.g. level sets. Manual intervention will also be possible at this stage. The output is then a list of refined features.

3.2 Hierarchy Generation

The majority of my existing work to date has focused on this first phase of the process. There are many different ways you can generate a partition hierarchy from an image, but the one I've found most interesting so far has been the waterfall transform from mathematical morphology. Its advantages are that it converges fast (i.e. it produces a relatively shallow hierarchy) and is free of any unnecessary parameters to tweak.

There are three stages to a hierarchy generation process using the waterfall, as shown in the following diagram:

$$\xrightarrow{\text{Image}} \text{PRE-PROCESSING} \xrightarrow{\text{Derived Images}} \text{WATERSHED} \xrightarrow{P_1} \text{WATERFALL} \xrightarrow{\text{Partition Tree}}$$

The waterfall itself generates the partition hierarchy, starting from an initial partition generated by a transform called the watershed. The input to the watershed is taken from a set of images derived from the original input image. Appendices B and C already go into some detail on how the latter two stages of this process work[2], so I won't repeat them here, but I would like to look at the details of the pre-processing stage.

Pre-Processing

As described in Chapter 2, one of the biggest difficulties with the watershed transform is the number of spurious local minima in the image, leading to an over-segmented resultant partition. One way of removing some of these minima is to smooth the image using a Gaussian filter (see Appendix C), but doing too much blanket smoothing of the image like this tends to blur the edges we are trying to find. My current approach does smooth the image a couple of times using a Gaussian filter, but then switches to using a more sophisticated approach, which I have called *Spatially-Variant Gaussian Filtering* (SVGF).

SVGF is really quite simple. It is based on the ideas behind non-linear diffusion filters, which try and smooth the image less in areas where we suspect there are edges, but it is a much less involved technique. I developed it primarily because diffusion filters can be intricate to implement and relatively slow to run, and I was interested in quick results in both senses.

The essence of the technique is to use a Gaussian filter whose standard deviation varies with its location in the image. High standard deviations cause greater smoothing, whilst a filter with a low standard deviation smoothes less strongly. The key is to ensure that the filter has a high standard deviation away from suspected edges, and a low standard deviation when near to them. This requires two things: an *estimate* of where the interested edges in the image lie (obviously we don't know where they are exactly: this is what we're trying to determine) and a mapping from the edge estimate image to the space of standard deviations (i.e. $\mathbb{R}^+$). It turns out that the gradient of the original image (after initial smoothing) provides a suitable edge estimate, but the mapping requires a bit more thought.

Taking a look at the 8-bit edge estimate image, we see that its grey levels range between 0 and 255, with higher numbers representing the higher estimated likelihood of an edge at that location. We're therefore looking for a function will map 0 to a high standard deviation (thus maximising the

[2]These are taken from articles I wrote for ACCU's Overload journal.

blurring away from the edges) and 255 to a low standard deviation (thus avoiding blurring the edges themselves). In between, we have a choice of functions, but it turns out that a sigmoid function works quite well. The canonical sigmoid function is given by

$$y = \frac{1}{1 + e^{-x}},$$

and we take that and transform it into a sigmoid function which starts high at $x = 0$ and rapidly drops off to a low level. To do this, we start by deciding on the points on the shape of the normal sigmoid curve to which we want 0 and 255 to map (my implementation maps $f_1 = 0$ to $x_1 = 6$ and $f_2 = 255$ to $x_2 = -100$). We then decide on the maximum and minimum values, y_h and y_ℓ, we wish our new sigmoid curve to take (my implementation used $y_h = 8$ and $y_\ell = 0$). Finally, we calculate the required function as follows:

$$\begin{aligned} s_x &= (f_2 - f_1)/(x_2 - x_1) \\ o_x &= f_1/s_x - x_1 \\ s_y &= y_h - y_\ell \\ o_y &= y_\ell \\ y &= s_y/(1 + \exp(-x/s_x + o_x)) + o_y \end{aligned}$$

The function I ended up using in my implementation was thus

$$y = \frac{8}{1 + \exp(106x/255 - 6)},$$

a function which starts off at nearly 8 when $x = 0$, then drops off rapidly to nearly zero at around $x = 30$ (see Figure 3.1). This turned out to work quite well in practice.

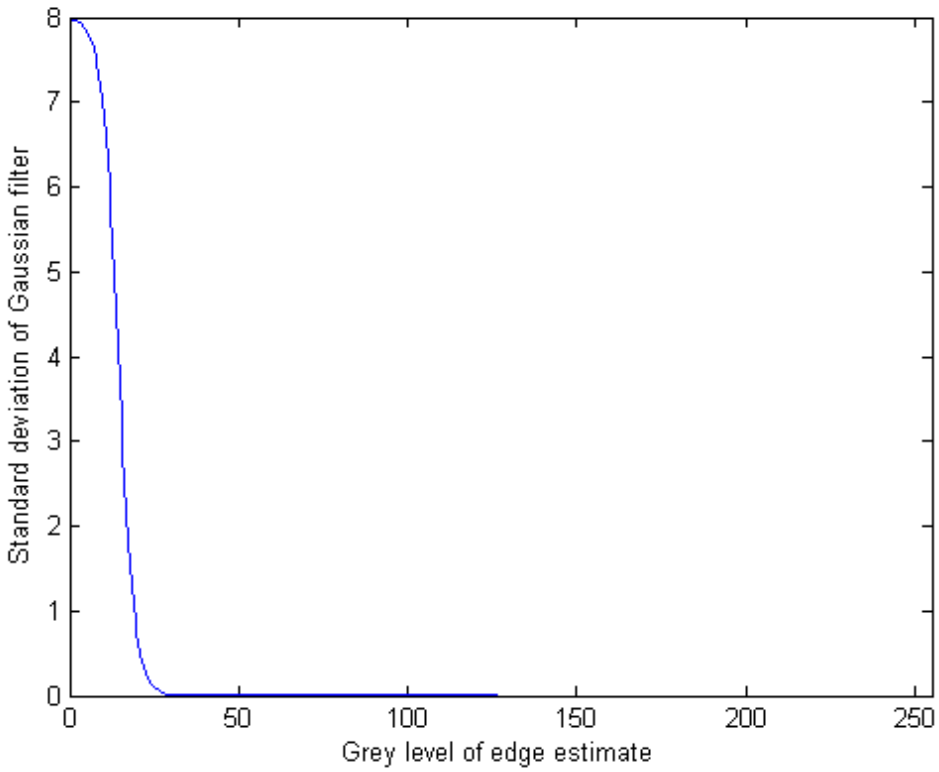


Figure 3.1: The sigmoid function which maps the edge estimate at a pixel to the standard deviation of the Gaussian filter to apply there

3.3 Partition Trees

As I have only just started thinking about the latter phases of my proposed segmentation algorithm, I can't describe them in any detail at this stage. However, I do want to describe partition trees and how I intend to use them, since they are integral to the whole process.

As Figure 3.2 shows, a partition tree is a rooted[3] tree with a region associated to each node. The region associated with each parent node is the union of those associated with its children. Furthermore, the regions associated with the nodes at any given level of the tree are mutually disjoint, and their union is the entirety of the image from which the tree was derived.

[3]It is also possible to deal with only a fixed number of levels at the bottom of the partition tree, for instance if we have no interest in merging regions until there is only one left. In that case we end up with a *partition forest*, but that's something which is beyond the scope of this report.

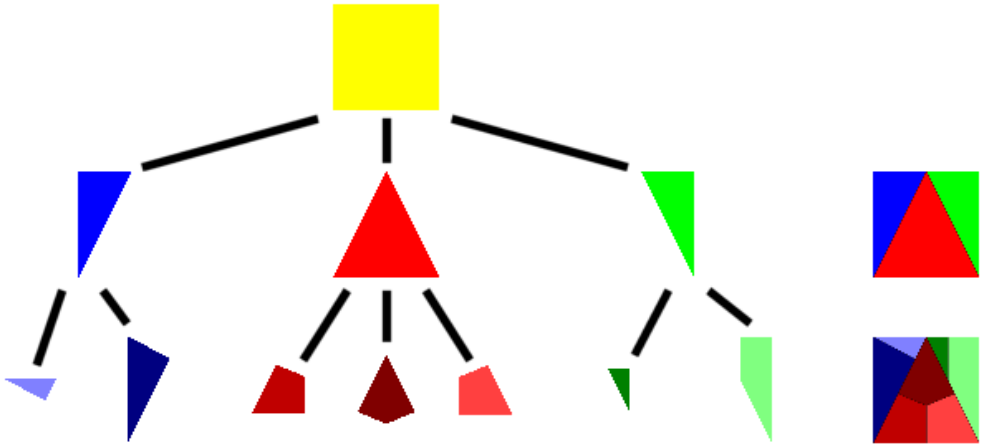


Figure 3.2: An example of a partition tree

When analysing the regions contained within a partition tree, we will want to associate properties with each node in the tree. I will refer to this process as *annotating* the tree. Partition trees are an ideal structure for this annotation task. The properties of parent regions often depend on their children (e.g. the area of a parent is the sum of the areas of its children), so by starting at the lowest level of the tree and working upwards, we can avoid recomputing existing results (see Figure 3.3).

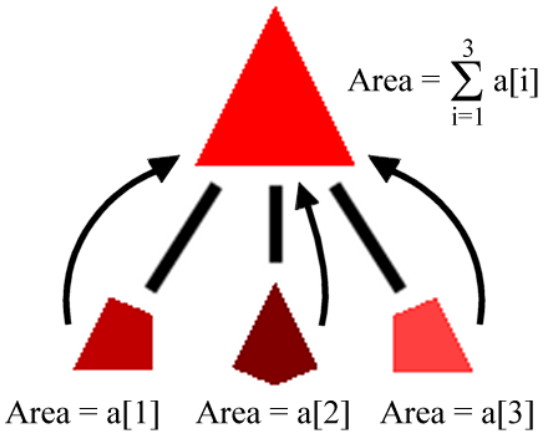


Figure 3.3: Annotating a partition tree from the bottom up

Partition trees are also a useful structure when it comes to the feature identification process. Identifying a feature simply involves removing the relevant node (and all its descendants) from the tree and regenerating all the regions associated with nodes above it on the path to the root of the tree (see Figure 3.4). It is thus a relatively efficient process.

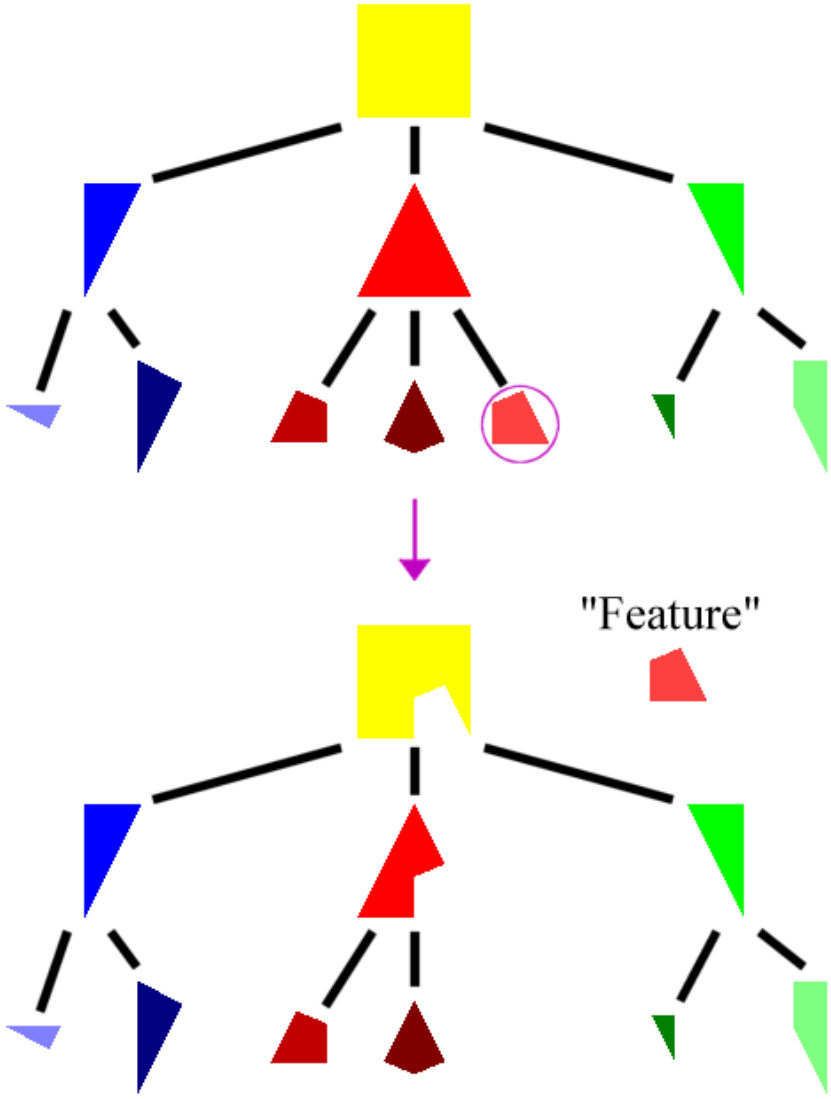
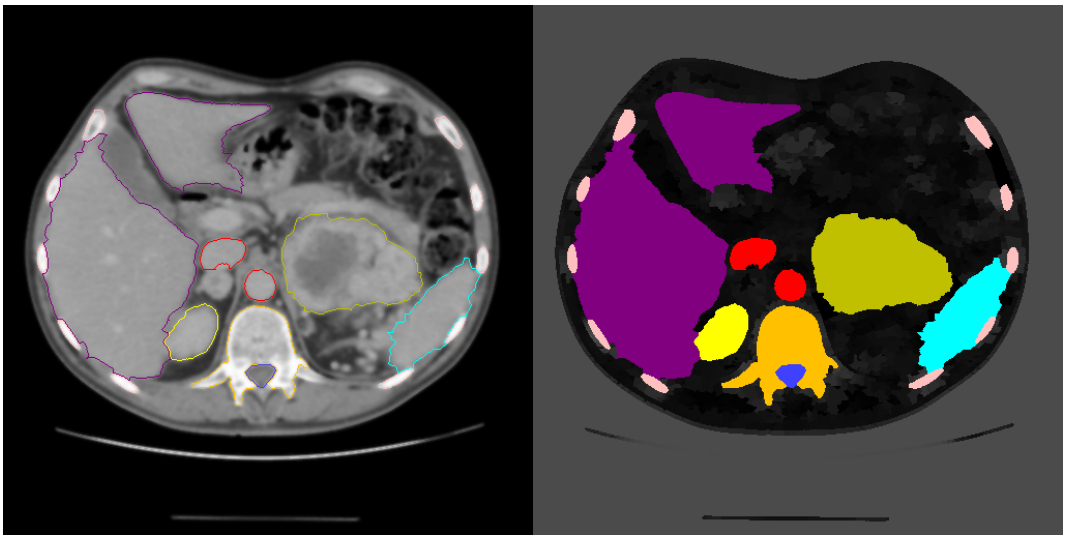


Figure 3.4: Identifying a feature in a partition tree: note that region properties will need to be regenerated for all the modified regions on the path to the root (not shown)

3.4 Results

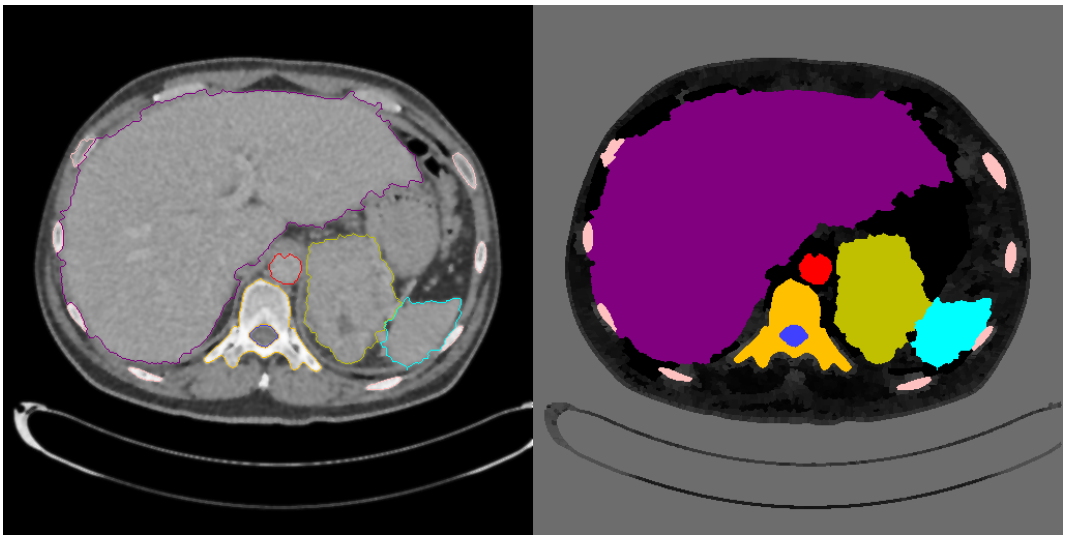
The results I currently have available are essentially the output of the hierarchy generation phase of my algorithm. On my new laptop, an image series of 255 images can be processed in under 5 minutes (roughly 1 second per slice). I have implemented interactive feature-marking functionality which allows the user to mark interesting features in the hierarchy for comparison to manually-generated 'gold standard' segmentations. The output looks promising (see the individual case studies below), but there is a lot of work still to be done.

Patient BT, Series 2, Slice 60 of 132, Region Marking



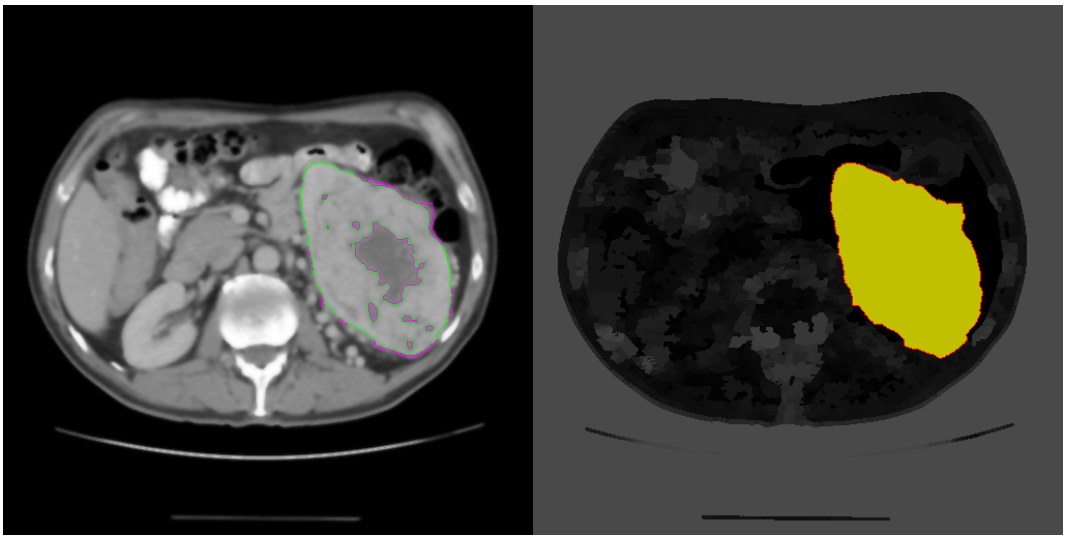
The key feature in this image is the large tumour on the patient's left kidney (marked in dark yellow). (Note that we are looking upwards at the patient lying on their back from the position of their feet.) Other visible features include the liver (purple), the right kidney (bright yellow), the aorta and inferior vena cava (red), the spleen (cyan), the ribs (pink), the spinal cord (blue) and the vertebra (orange). Series 2 for each of the data sets available was of significantly higher quality than the corresponding Series 3, so the features were easy to interactively identify.

Patient MC, Series 3, Slice 119 of 255, Region Marking



In this image, a tumour can also be seen on the left kidney. The general image quality is seen to be poorer (this series was taken at a lower resolution), making life harder for the watershed algorithm by introducing vast numbers of spurious regional minima. The image shown is post-smoothing: the original image was much worse. A reasonable amount of manual editing was required to obtain this result.

Patient BT, Series 2, Slice 70 of 132, Necrosis Analysis



This final case study illustrates my initial attempts at analysing tumour necrosis. At this stage, a very basic thresholding algorithm is being used on an already-identified tumour region; the threshold was set empirically. Areas surrounded by a pink border (darker grey on the scan itself) are speculated to be necrotic; those surrounded by a green border are speculated to be non-necrotic. The results are interesting, but much more work is needed on this. One point worth noting is that small areas round the edge of the tumour have been identified as necrotic using this scheme. This is clearly not the case, since necrosis manifests itself within a tumour, but it does give some insight into where the desired tumour boundary should lie. In particular, the green border within the tumour boundary looks like a much better approximation than the existing tumour boundary. This is definitely worth looking into.

3.5 Conclusions

There is a great deal of work still to be done on this, but the initial results seem relatively promising. More work needs to be done on hierarchy generation for grainy images, but in the short term it makes more sense to focus on property generation and feature identification. I expect partition trees to be extremely useful in this regard.

Whilst working on segmentation is inherently interesting, it should always be borne in mind that my method is ultimately intended to be put to use for higher-level applications, e.g. analysis of tumour necrosis. My current plan is to develop the two concurrently, since the requirements for necrotic analysis will determine the quality of output necessary for my segmentation method.

Chapter 4

Plan of Action

4.1 Goals

I have two main goals for my doctorate:

- To develop a clinically-useful tool to quantify the extent of necrosis in renal tumours. (Note that this is restricted in its scope, making it more likely to be feasible in the time-frame.)
- To develop a geometric growth model for kidney tumours based on existing work done in Oxford on tumours of the liver [26].

Both of these goals necessitate the segmentation of tumours from abdominal scans, which has been the main focus of my medical imaging work to date. I expect my doctoral thesis to be a combined work on segmentation and geometric growth modelling.

4.2 Proposed Route to Goals

I have two years of doctoral funding remaining, in the form of an EPSRC Doctoral Training Award. My current intention (which is obviously subject to later modification as my doctorate progresses) is to spend that time as follows:

Dec 07 - Mar 08	Region generation for segmentation: watershed and waterfall
Mar 08 - Jun 08	Literature survey on tumour necrosis Requirements analysis with medics Resubmission of transfer report
Jun 08 - Sep 08	Quantification of tumour necrosis and validation with radiologist Organ identification for segmentation
Sep 08 - Dec 08	Quantification of tumour necrosis and validation with radiologist Geometric growth modelling
Dec 08 - Mar 09	Refinement of clinical tool Organ identification for segmentation
Mar 09 - Jun 09	Refinement of clinical tool Geometric growth modelling Organ identification for segmentation
Jun 09 - Sep 09	Formal validation of clinical tool Geometric growth modelling
Sep 09 - Dec 09	Geometric growth modelling Organ identification for segmentation Writing up
Dec 09 - Mar 10	Writing up

The colour-coding indicates the various different strands of work involved. Green items relate to the clinical tool I plan to develop, blue items relate to my segmentation work, and purple items relate to geometric modelling. Finally, red items relate to report and thesis-writing.

4.3 Feasibility

I believe that both of my goals are achievable, by me, in the time involved. The clinical tool I wish to build is clearly feasible: I have already demonstrated in the previous chapter that, even at this stage, I can obtain a rough segmentation of the necrotic areas of a tumour using my software. Over the coming year, I will continue refining my segmentation process to better determine the tumour and its necrotic areas on each slice. I hope this can be reasonably automated, although evidently it will be important to allow the radiologist to check and refine the results. It is worth mentioning that my work is not dependent on fully-automating the segmentation process, which would be a hard problem to solve.

Whilst I hope to make it as automated as possible (an idea I have discussed with the radiologist, who agrees that it is desirable provided the quality of the results is not compromised), I can fall back to using a less-automated process if I encounter problems. In either case, the clinical tool will be implementable in the time-frame allowed.

My work on geometric modelling of renal tumour growth is also thoroughly feasible. It is likely to be an incremental step from existing work done on liver tumours in Oxford, thus limiting the risk involved. Nevertheless, it is an exciting and, as yet, relatively unexplored area of research. It is important to note that the segmentation requirements for geometric modelling of tumour growth are much easier to meet than those for modelling at the cellular level: small variations in a segmentation lead to similarly small variations in the geometry.

4.4 Usefulness to Clinicians

Both goals of my work are potentially useful to clinicians. The clinical tool is the more obviously useful, in the sense that it is based exclusively on what the medics are telling us they need: a program that will help them quantify the extent of necrosis in tumours more accurately. However, as the existing work on liver tumours shows, models of tumour growth are also potentially applicable in a clinical setting, as they can help clinicians determine a tumour's response to therapeutic intervention and hence decide on the best course of action.

4.5 Validation of Results

We have had initial discussions with Zoe Traill (the radiologist) about how to validate the results of my segmentation algorithm and tumour necrosis quantification. In terms of the segmentation algorithm, she has indicated that she would be happy to manually segment images for us. The results of my program can thus be compared to her 'gold-standard' segmentation, and evaluated in terms of (for example) the number/percentage of pixels misclassified.

In terms of tumour necrosis quantification, her suggestion is to do a three-way comparison between the results obtained by the pathologist, the results derived from the radiological scans, and the results my program will ultimately produce. Necrotic analysis from the scans currently involves (essentially) putting an axis-aligned box around the area of necrosis in question: the existing results

are thus relatively coarse, and any validation procedure will have to be carefully designed with this in mind.

It isn't possible to say at this stage how I intend to validate my work on tumour growth, since the details of what it will ultimately be used for (and hence the details of what results it should produce) have yet to be determined.

4.6 Fall-Back Plans

It is not expected that a fall-back plan will be used, as our discussions with the medics on the potential for collaboration have been very positive: indeed, following a recent meeting, they have expressed an interested in holding regular meetings with us about my topic of research in future. We believe, however, that even were this not the case, there would be enough work on automatic segmentation, with the existing data I have available, to form the basis for my doctorate.

Appendix A

Real-Time Dynamics Simulation of Long Hair over Arbitrary Meshes

Note: The text of this chapter is taken (largely) verbatim from the paper of the same name that we submitted to Eurographics 2008 [19].

A.1 Introduction

Simulating long hair dynamics in real-time remains a difficult problem for researchers in physically-based modelling, primarily because of the large numbers of hairs needed to attain a desirable level of realism. A number of different approaches have been developed to help deal with this scale problem. For example, [71] describes a level-of-detail approach which dynamically switches between rendering hairs as low-resolution strips, medium-resolution clusters or high-resolution strands, based on factors including viewing distance, visibility and hair motion, and [24] presents a real-time approach which combines neighbouring hairs into wisps.

Dealing with the large number of hairs required is only part of the challenge, however. Even at the level of individual hairs, there are a number of problems to be overcome: among others, how to simulate the dynamics of a hair as it is acted upon by external forces caused by (for example) gravity and wind, how to handle collisions between hairs and their environment, and whether and how to try and handle collisions between the hairs themselves.

Solutions to these problems tend to be summarised in the literature rather than described in full. The intent of this paper, therefore, is to focus in detail on how to implement hair dynamics using mass-springs, and how to apply an appropriate collision detection and response (CDR) method to handle hair-head collisions. We have not attempted to deal with hair-hair collisions, as explicit hair

models like ours are not well-suited to handle them, but past approaches tried by others have included modelling the volume of hair as a continuum [21] and using a thin-shell volume method [27]. Interestingly, the latter approach also facilitates hair combing.

It should be noted that using a mass-spring approach is not inherently novel; for example, see [41], [64], [61]. Rather we focus on the design of the underlying physics system and the development of a fast CDR method for resolving collisions between long hairs and arbitrary meshes.

In the rest of this paper, §A.2 describes the physics system used, §A.3 the way that we model hairs, §A.4 the CDR system and §A.5 the rendering method used. Results are summarised in §A.6, which is followed by our conclusions.

A.2 Physics

In order to simulate long hair dynamics, we developed a generic physics system which allows users to define and instantiate any number of different types of object, and connect them (using springs) in an intuitive manner. Since constructing such a system is interesting, non-trivial and key to the hair modelling specifics which follow, the following subsections explain how it works in detail.

It is important to note at this point that our implementation separates simple physical simulation (how to update the physical objects over time, without worrying about collisions) from collision resolution, which is performed as a separate (later) stage at each frame. The reasoning behind this is that collision resolution is dependent on the specifics of the application, whereas physical simulation can be generic. The details of collision handling are accordingly deferred until later in the paper.

A.2.1 Objects and Forces

The software of the physics system maintains a set of physical objects (without knowing their individual types), together with a numerical method for each object which is used to update it over time. Each type of object has its own numerical method, as well as a (finite) set of key points to which forces can be applied (see Figure A.1).

In our system, it is important to distinguish between the related notions of a *force* and its *value*. For instance, an object may have a force exerted on it as a result of being connected to another object with a spring, but the value of that force depends on the relative position of the other object. That

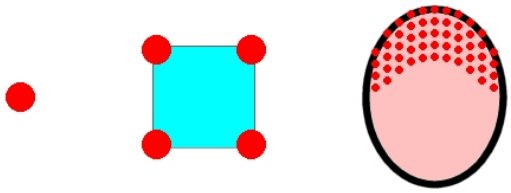


Figure A.1: Point masses (left) have only one key point (at their own position), whereas rigid bodies (centre) can have any number of them. We can model a head (right) as a rigid body with a key point for each hair root, as will be seen later.

is, the value of the force depends on the *context* in which it is being evaluated. We thus explicitly introduce the notion of a physics context as capturing the properties (e.g. position, velocity, etc.) of all the objects in the system at a given point in time, and define a force as being a function taking such a context and yielding a value in $\mathbb{R}^3$.

A.2.2 Force Appliers

Having satisfactorily defined forces in our system, we can now introduce the higher-level concept of *force appliers*. These serve two important purposes: they allow forces to be applied to objects in our system in an intuitive manner, and they store details about themselves to allow the forces to be evaluated later. A *linear spring* (see Figure A.2), for example, might store its Hooke constant and initial length, as well as handles to the two key points it connects. These handles can then be used to query the current physics context for the positions of the key points when evaluating the forces.

In terms of the actual code, connecting a linear spring simply involves specifying its properties and then calling a method to connect it; the parameters to this method are the object handles and key point indices of the two key points the spring should connect:

```
LinearSpring s = new LinearSpring(k, ℓ0);
s.connect(obj1, keypt1, obj2, keypt2);
```

In practice, there are any number of different possible force appliers, of which linear springs are only one example. In our simulation, we also made use of gravity, wind, (viscous) air resistance and *angular springs*. The first three force appliers all store a handle to the object to which they are applied, which they use to look up its properties in a given context before calculating a force value.

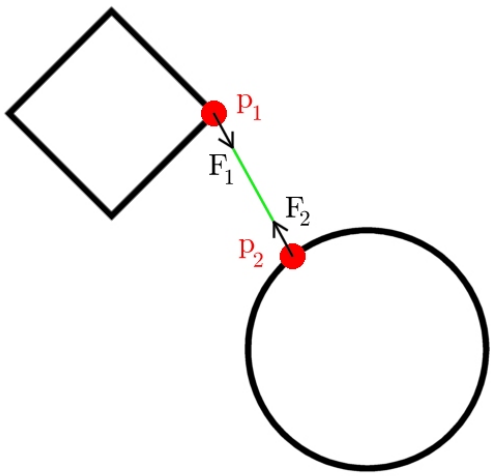


Figure A.2: Connecting key points (red) on two rigid bodies with a linear spring (green): as a result, a force $\mathbf{F_1}$ is applied to the square and a force $\mathbf{F_2}$ is applied to the circle. Force $\mathbf{F_1}$ points in the direction $\mathbf{p_2} - \mathbf{p_1}$ and has magnitude $k|\ell - \ell_0|$, where $k > 0$ is the spring's Hooke constant, ℓ_0 is its initial length and $\ell = |\mathbf{p_2} - \mathbf{p_1}|$ is its current length. Force $\mathbf{F_2}$ points in the opposite direction and has the same magnitude.

In particular, the gravity force applier looks up the object's mass and scales it by $(0, 0, -g)$, where $g \approx 9.81$ N/kg, the wind force applier looks up the object's mass and scales it by a wind vector $\mathbf{w}$, and the air resistance force applier looks up the object's velocity and scales it using $\mathbf{F} = -b\mathbf{v}$.

In the interests of placing them in their proper context, we defer discussion of the last force applier, angular springs, until Section A.3.4.

A.2.3 Numerical Methods

Each object in the physics system has a numerical method associated with it to update its properties (e.g. position, velocity, etc.) over time. The numerical method used depends on the type of the object, not on the object itself, so that (for example) all point masses are updated using the same numerical method.

For our implementation, we used a two-stage predictor-corrector method to obtain good results. This was chosen as a trade-off between forward Euler, which would have been unstable and inaccurate, and a Runge-Kutta method, which would have been slow because of the interdependencies between the objects in our system. In practice, the approach we used worked quite well.

An overview of our method can be seen in Figure A.3. A formal presentation of the two numerical stages will help to clarify how it works. Let m_i be the mass of object i and $\mathbf{p_i}(t)$, $\mathbf{v_i}(t)$ and $\mathbf{a_i}(t)$ be its

true position, velocity and acceleration (respectively) at time t. If the simulation time-step is Δt, then let $P_i^k \approx \mathbf{p_i}(k\Delta t)$, $V_i^k \approx \mathbf{v_i}(k\Delta t)$ and $A_i^k \approx \mathbf{a_i}(k\Delta t)$ be the numerical approximations to the above at time $k\Delta t$, i.e. after k time-steps have elapsed. Let $\mathbf{f_i}(t) = \mathbf{f_i}(m_i, \mathbf{p_{1\ldots n}}(t), \mathbf{v_{1\ldots n}}(t))$ be the function which calculates the net force on object i at time t, and $F_i^k = \mathbf{f_i}(m_i, P_{1\ldots n}^k, V_{1\ldots n}^k)$ be the approximation to it at time $k\Delta t$ (whence of course $\mathbf{a_i}(t) = \mathbf{f_i}(t)/m_i$ and $A_i^k = F_i^k/m_i$). The predictor method is then:

$$\begin{aligned} P_i^{(p)} &= P_i^k + V_i^k \Delta t \\ V_i^{(p)} &= V_i^k + A_i^k \Delta t \\ A_i^{(p)} &= \mathbf{f_i}(m_i, P_{1\ldots n}^{(p)}, V_{1\ldots n}^{(p)})/m_i \end{aligned}$$

And the corrector method is:

$$\begin{aligned} P_i^{k+1} &= P_i^k + \frac{\Delta t}{2}(V_i^k + V_i^{(p)}) \\ V_i^{k+1} &= V_i^k + \frac{\Delta t}{2}(A_i^k + A_i^{(p)}) \end{aligned}$$

It is worth noting that in our formulation, each predicted force potentially depends on the predicted positions and velocities of all the objects in the system, which makes the simulation very difficult to parallelize. Whilst some interdependence between objects does make sense (consider the ubiquitous linear spring example), in most cases each object will only depend on a very limited subset of the objects around it. Considering the objects as an undirected graph, therefore, with arcs only between those objects that are connected by a spring in the simulation, we can observe that the objects in each connected component of the graph can be processed separately.

A.3 Hair Simulation

Having discussed how to implement an appropriate physics system for hair simulation, we can now turn our attentions to the problem of hair itself. There are a number of issues to consider: how to implement a base to which to attach the hairs, how to distribute the hairs over that base and allow their properties to be specified, and finally how to simulate their dynamics.

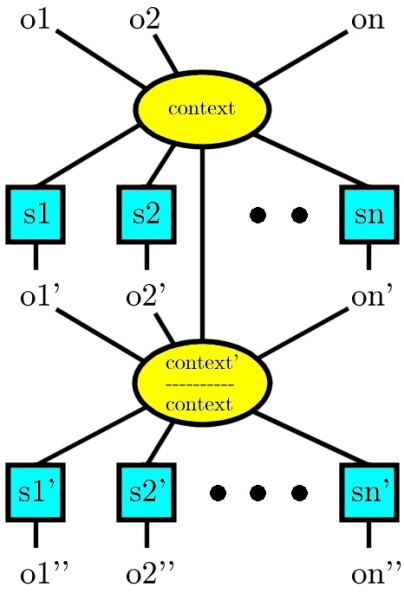


Figure A.3: Updating n objects over time using a two-stage predictor-corrector method. Each object o_i has a predictor stage s_i and a corrector stage s_i' associated with it. The initial objects $o_1, \ldots, o_n$ are combined into a physics context, $\mathbf{context}$, which is fed into each of the predictor stages. Each predictor stage s_i yields a predicted update o_i' of object o_i after the time-step. These predicted objects are themselves combined into another context, $\mathbf{context}'$, and the two contexts are both fed into each of the corrector stages. Each corrector stage s_i' then yields the true update o_i'' of object o_i, by combining the objects from both contexts for a more accurate update.

A.3.1 Hair Bases

Before we can simulate hairs, we need a *hair base* to which to attach them. In our initial implementation, we attached hairs to an ellipsoidal head (as in Figure A.1), but we have since extended this to allow hairs to be attached to arbitrary meshes. In our physics system, a hair base is represented as a rigid body whose key points are potential hair roots (see Figure A.4). To attach a hair to it, we simply connect the end of a hair to its root on the hair base with a spring.

A.3.2 Hair Properties

Each hair has a number of properties associated with it, including its length, colour scheme and *inclination* (the last of these will be explained shortly). In addition, there are more general properties which vary over the surface of the hair base, e.g. hair number density. Our implementation permits the definition of any number of hair regions, each of which is associated with a given set of hair properties (length, colour scheme and number density). We used a material-based method to do this, applying a different material to each region of the mesh where we wanted hair (see Figure A.15).

Figure A.4: Hairs are attached to a hair base by connecting their ends to key points on the hair base (shown in red) with springs. The penguin model was created in Blender using the tutorial at [7].

Hair number density is defined to be a small positive integer for mesh triangles in each region marked with a special material. In regions which should be hair-free, it is defined to be zero. Hair colour schemes will be discussed in more detail in Section A.5. A hair's inclination is not specified by the user; rather, it is defined to be the normal to the mesh triangle at the hair's root, and serves two purposes. On the one hand, we use it as the starting direction of the hair (in our simulation, the hairs initially point upwards at an angle and are allowed to fall over the hair base under the influence of external forces like gravity); on the other, it proves useful as a gradient endpoint constraint when we come to render the hair as a cubic spline (again, see Section A.5).

Whilst our implementation allows us to vary hair properties on a per-region basis, it doesn't give us fine-grained control over hair length and colour. Currently, hair regions are specified interactively in Blender via the use of 'special' materials, with hair properties for each region subsequently being defined procedurally in the code. This approach limits the amount of information that the user of the modelling program (the 'artist') can provide to the code about the desired appearance of the hair, as well as introducing a time lag before the artist can see how his mental picture of the hair design looks in reality. The best solution to this problem would be to build an interactive hair application system. Interactive approaches (such as that in [77]) give us the fine-grained control we seek, as well as allowing the artist to visualise the end results in real-time.

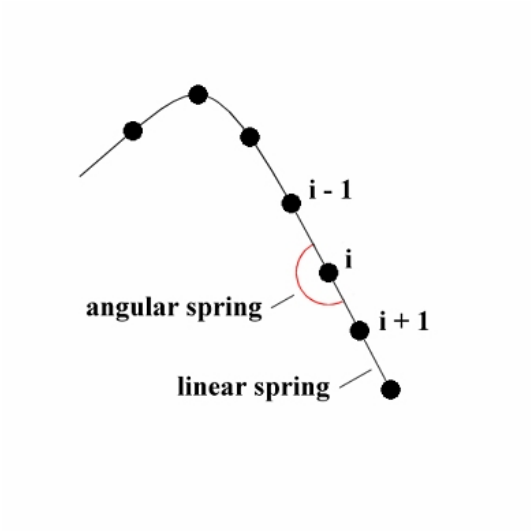


Figure A.5: We represent hairs as chains of point masses connected by linear and angular springs.

A.3.3 Hair Distribution

To distribute the hairs over the mesh, a number of hair roots were (pseudo-)randomly generated in each mesh triangle based on that triangle's hair number density (as specified previously), weighted by the area of the triangle so that larger triangles received more hair roots.

A.3.4 Hair Dynamics

Simulating the dynamics of hairs involves modelling them in terms of lower-level physics primitives. In our implementation, this meant representing hairs as chains of point masses connected by springs (see Figure A.5). Each mass in the chain (except those at the ends) is attached to those above and below it by linear springs. The top-most mass is connected to the root on the hair base.

In addition to connecting the masses in the chain with linear springs, we used angular springs in our implementation to resist unrealistic bending of the hair. As illustrated in Figure A.6, applying an angular spring at the i^{th} point in the chain involves exerting forces on points $i-1$ and $i+1$ which have a restoring effect on the angle at point i. Specifically, the forces try and restore the angle θ at point i to a user-specified θ_{rest} angle. As can be seen from the equation given in the figure, the strength of the forces increases quadratically as the difference between θ and θ_{rest} increases. The overall strength of the forces applied is governed by a user-specified constant, κ.

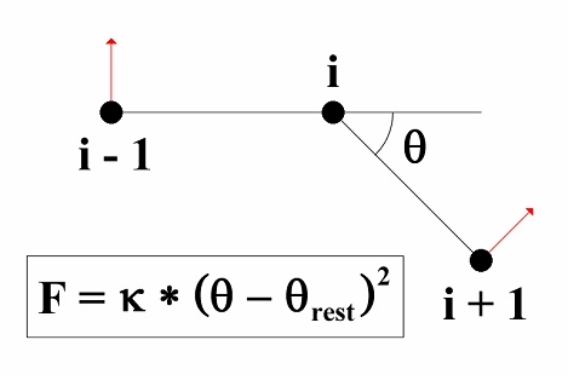


Figure A.6: Angular springs can be applied to three consecutive points in a chain to resist bending: they apply forces to the outer two points to try and restore the angle θ at the middle point to a user-specified θ_{rest} angle, which is generally zero.

A.4 Collisions

As previously mentioned, collision handling in our implementation was performed separately from physical simulation. We chose to ignore hair-hair interactions, both because our explicit model was ill-equipped to handle them, and because they tend to be largely invisible to the viewer. We focused instead on handling collisions between the hairs and their base.

When attempting to simulate hair in real-time, runtime speed is of vital importance. Our implementation therefore used a simple point-based collision method. We also used a simplified mesh for collision handling, since the high-detail geometry designed for rendering would have impeded performance.

Our algorithm resolved each hair in turn against a *collision mesh* (see Figure A.7), an expanded, simplified version of the mesh used for rendering. There were a number of steps involved in this procedure:

1. Starting from the point mass on the hair closest to the hair root, each mass was checked to see whether it lay inside the collision mesh. This was done by building a binary space partitioning (BSP) tree from the mesh. BSP trees were introduced in [16] and can be used to classify volumes of space as being either solid or empty. They are constructed by recursively splitting space across a plane until each leaf of the tree can be uniquely classified as one of the above (see Figure A.8). There is then a simple recursive algorithm to decide whether a point is in solid or empty space.

2. If a point mass was found to lie in solid space (i.e. inside the mesh), it was projected back onto

the surface. The projection was done by casting a ray back towards the surface (in a direction which will be discussed shortly) and finding the point of intersection using another BSP tree algorithm called *find first transition*. Given the ray $\mathbf{r}(\lambda) = \mathbf{start} + \lambda(\mathbf{dir})$, this algorithm, shown in Listing 1, returns the point closest to $\mathbf{start}$, if any, at which the ray transitions from solid to empty space (or vice-versa). It essentially works in a similar way to a line segment classifier, but it prioritises the side of the tree nearest the start of the ray. Starting from the root of the tree, the algorithm classifies the ray against the split plane at the current node. If it is entirely on one side of the plane, it is passed down the relevant side of the tree. If it straddles the plane, it is split in two at the plane and the two halves are passed down their respective sides of the tree, with the half nearest the start of the ray being processed first and the other half only being processed if no transition point has been found in the first half. Finally, there are a number of coplanar special cases which are documented implicitly in the code itself. The direction in which the ray should be cast is non-obvious. The negation of the point mass's velocity (which we will denote $-\mathbf{v} = (-v_x, -v_y, -v_z)$ in what follows) would seem to be a natural choice, but was found to work poorly in practice. The approach we eventually used was to generate a direction set D and pick the direction which resulted in the least movement of the point mass. The directions used could be varied; we used $D = \{(-v_x, -v_y, 0), (-v_x, 0, -v_z), (0, -v_y, -v_z)\}$.

3. Having moved the point mass to the surface of the mesh, our final step was to apply a length constraint to the hair segment joining the moved mass to its predecessor in the chain (see Figure A.9). This was important because it prevented unwanted spring forces from being generated by stretching the hair. The length constraint restores the hair segment to the length it had before the movement by moving the (moved) mass along the direction of the hair segment. Whilst this runs the risk of moving the mass back inside the collision mesh by a small amount, this is acceptable in practice since we are only really interested in keeping the masses (and hence the hair on which they lie) outside the inner mesh we use for rendering. Provided we sufficiently expand the rendering mesh when generating our collision mesh, this tends to be the case.

It is worth noting that the implemented collision-handling scheme is rather limited (its attractions are its simplicity and speed). By using a point-based approach (i.e. testing individual point masses against the mesh), we rely on the points on a hair being sufficiently tightly-spaced that all its collisions with the hair base will be detected.

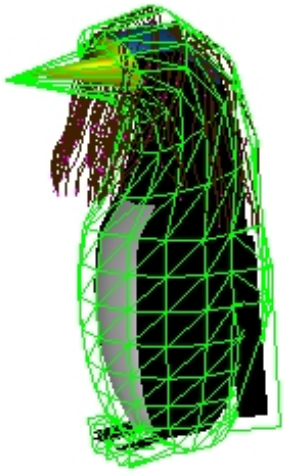


Figure A.7: The mesh we used for collision handling was an expanded, simplified version of that we used for rendering. It was constructed in Blender, using its Shrink/Fatten Along Normals function.

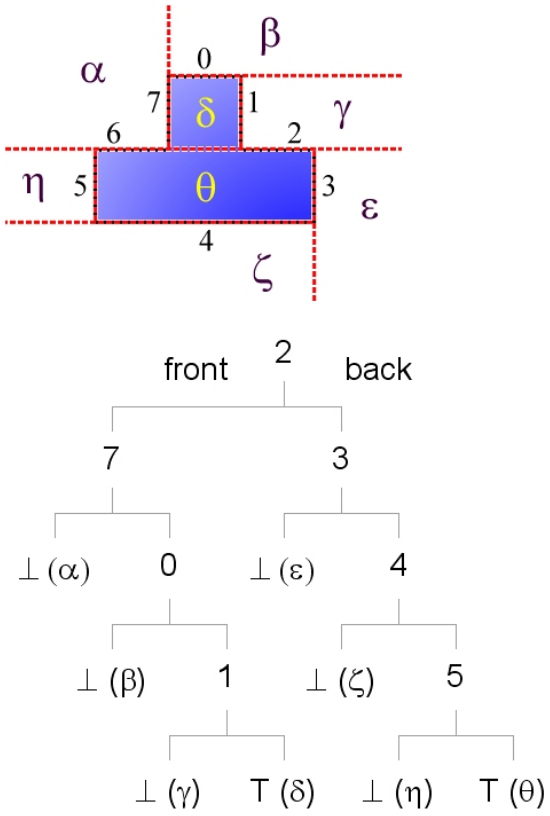


Figure A.8: A BSP tree divides space into two recursively, such that the splitting plane at a node divides everything in the node's left subtree from everything in its right subtree. The tree can be used to classify space as either solid ($\top$) or empty ($\bot$). For a detailed discussion of BSP tree construction, including sample code, see Appendix D of [20].

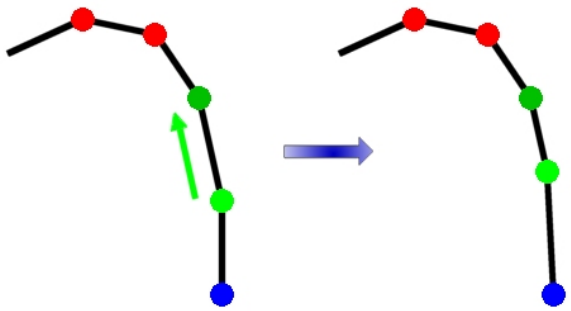


Figure A.9: Having moved an embedded point mass (light green) back to the surface of the mesh, we ensure that the hair segment joining it to its predecessor (dark green) remains the same length as it was before by moving the mass along the segment. This ensures that no unwanted spring forces are introduced. We then proceed to process the next point in the chain (blue) in the same way.

A.5 Rendering

A.5.1 Main Scene

Although the primary focus of our work was on the dynamics of hair, how to render the results remained important. The scene was rendered using Phong shading, implemented as shaders written in the OpenGL Shading Language (GLSL). The hairs were rendered as cubic B-splines.

These splines were chosen in preference to non-uniform rational B-splines (NURBS) because they were adequate for the task and the algorithm used to make them interpolate a given set of points (see [69] and [20] for further details) is faster and requires fewer user-specified constraints than that used for NURBS. (Indeed, the use of NURBS would have been a hindrance rather than a help, since the extra flexibility in curve design it would have given us was overkill for the task at hand.) We used the interpolation algorithm to interpolate the point masses on each hair, specifying the gradients at the ends of the hair as our constraints. The gradient at the end nearest the hair base was specified to be the hair inclination (as mentioned earlier); that at the other end was specified to be the down vector, $(0, 0, -1)$.

Since spline interpolation is quite a costly process, it was important to find ways to minimise the time it took each frame, in order to give as much processor time as possible to the dynamics simulation. Notationally representing a cubic B-spline as a weighted infinite sum of cubic basis functions, $\mathbf{C}(t) = \sum_{a=-\infty}^{\infty} \mathbf{d_a} N_a^{(3)}(t)$, the key step of the spline interpolation algorithm involves finding the weights $\mathbf{d_a}$ by solving a matrix equation of the form $M\mathbf{d} = \mathbf{y}$, where $\mathbf{y}$ specifies the

Figure A.10: Rendering hair as cubic B-splines

points to be interpolated, $\mathbf{d}$ represents the unknown weights and the equation as a whole expresses the relationship between the weights, the points to be interpolated and the constraints. Solving this linear system takes a time cubic in the dimensions of the matrix; for that reason, we prefer to do it only once, at the start of the simulation. We achieve this by guaranteeing that the constraints (the gradients at the ends of the hair) will not change over time, thus fixing M in the equation and allowing us to calculate and cache M^{-1} for each hair on initialisation. Our system can then directly calculate $\mathbf{d} = M^{-1}\mathbf{y}$ for each hair as the positions of its set of point masses ($\mathbf{y}$) varies over time, a far more efficient (quadratic) process.

When rendering the splines (as a number of connected line segments), it was important to allow different colour schemes to be applied. To do this, each hair was assigned a *colourer*, a function from its spline's t parameter to RGB-space. This was repeatedly evaluated when rendering the line segments and used to change the hair's colour along its length. We experimented with two different types of colourer – one which assigned a single colour to the whole hair (thus using a flat shading model) and one which linearly interpolated from one colour at the hair root to a different colour at the other end – but any function of the right type would work in practice. The results of our spline-based hair rendering can be seen in Figure A.10 (there are further examples at the end of the paper).

A.5.2 Camera Control

To make it easy to view our hair simulation from any angle, we implemented a number of different *camera controllers*. The camera itself was represented in terms of its position $\mathbf{p}$ and local coordinate system $(\mathbf{n}, \mathbf{u}, \mathbf{v})$, and the `gluLookAt` function (from the OpenGL utility library GLU) was used to change the OpenGL state appropriately whenever it moved. Each controller then changed the position of the camera based on either user input or time.

Keyboard camera control was relatively simple given our camera representation (e.g. moving forwards simply involved moving the camera's position in the direction of its look vector, $\mathbf{n}$). Of more interest is our time-based camera controller, which we based once again on cubic B-splines. The idea was as follows: given m (position, look) pairs $(p_1, \ell_1), \ldots, (p_m, \ell_m)$ specifying the desired positions and look vectors of the camera over time, we use a spline $\mathbf{P}(s)$ to interpolate the positions and another $\mathbf{L}(s)$ to interpolate the look vectors. We then specify that the up vector, $\mathbf{v}$, for the camera is the global up vector $(0, 0, 1)$ and that $\mathbf{u}$, the side direction for the camera, is $\mathbf{v} \times \mathbf{n}$. (This means that we can't specify the global up vector as the look vector for the camera, of course, but that wasn't a huge problem in our case.) Finally, we make the spline parameter s a function of the simulation time t, e.g. $s(t) = kt$ for some constant speed k, and set the camera's position and look vectors at time t to $\mathbf{P}(s(t))$ and $\mathbf{L}(s(t))$.

Using this scheme, it was easy to make the camera follow a path around the model used for hair simulation, as seen in Figure A.17.

A.6 Results

The performance of our simulation (in frames per second), for various different parameter configurations, is shown in Table A.1. Whilst only limited conclusions can be drawn from such a small data set (especially since it was observed that the timings did vary slightly if the tests were repeated: the values given are an average), the results are suggestive. As can be seen in Figure A.11, as the number of hairs increases, the time taken to process each frame appears to grow linearly. This is in agreement with our expectations, since all the key algorithms we are using are linear. For small numbers of hairs, the time taken per frame levels out at around 50ms. This is an artifact of our requesting simulation updates at regular 50ms intervals, and as such is not significant. It is worth observing that we would

expect the time taken to level out at some point in any case, since as the number of hairs decreases, other aspects of the simulation (e.g. rendering the mesh) begin to dominate.

Model	Hairs	Point masses / hair	FPS
Penguin	24	4	19.7
	48	4	18.3
	96	4	15.2
	192	4	8.4
	384	4	4.9
	768	4	2.7
	96	6	12.6
	96	8	10.0
Horse	376	4	5.2
Gnu	208	4	8.7

Table A.1: This table shows how the simulation performance (measured in frames per second) varies with the model used, the number of hairs simulated and the number of point masses used per hair. (Timings obtained on a single 2.4GHz Pentium 4 CPU with an NVIDIA GeForce FX Go5600 GPU.)

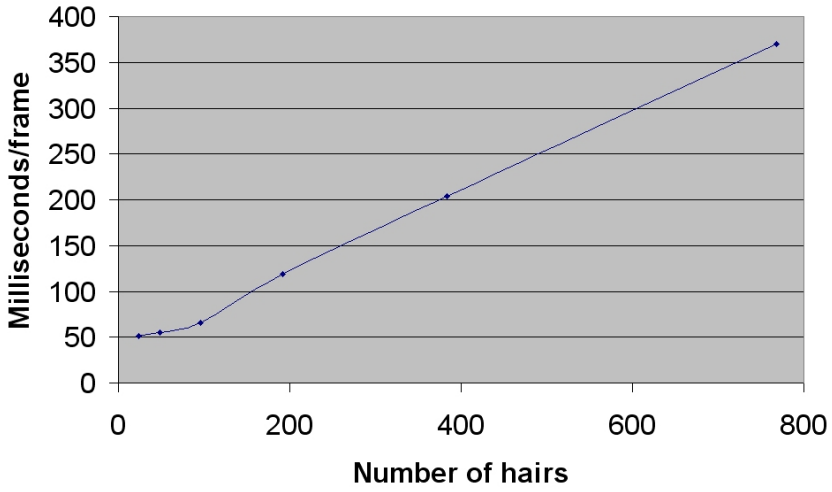


Figure A.11: As the number of hairs (on the penguin model) increases, the frame time increases linearly.

For any significant number of hairs, the time taken to render the mesh had little effect on the frame rate. In particular, the time taken to render 384 hairs on the penguin mesh (508 triangles) and that taken to render 376 hairs on the horse mesh (4944 triangles) were strikingly similar, as were the times taken to render 192 hairs on the penguin mesh and 208 hairs on the gnu mesh (6224 triangles).

A.7 Conclusions

We have described a method of explicitly modelling hair dynamics using mass-springs, together with some details of the implementation. Most of the ideas used are not radically different from those seen elsewhere; rather, the emphasis has been on providing an implementation constructed using good software engineering principles and making good use of the GPU for display purposes. With this in mind, the implementation provides real-time performance, and is easily ported to different architectures. For example, we have successfully ported the code to an IBM simulator for the Cell processor.

There are a number of aspects of our approach which have the potential for some interesting future work. At present, our implementation focuses primarily on hair dynamics, at the expense of other aspects of the hair simulation problem such as hair shape modelling [39]. In addition to modelling straight, dry hair, it would be interesting to try and apply similar ideas to other types of hair (e.g. curly hair or wet hair [70]). Another useful extension would be to implement a level-of-detail approach to allow more hair to be simulated.

Listing 1 The find first transition algorithm for BSP trees.

```
FIND-TRANS(start, dir, tree)
  LARGE_CONSTANT ← 10000
  end ← LARGE_CONSTANT * dir + start
  root ← ROOT(tree)
  (_,pt) ← FIND-TRANS-S(start, end, root)
  return pt

FIND-TRANS-S(start, end, node)
  if IS-LEAF(node) then
    if IS-SOLID(node) then
      return (CT_INSIDE, NULL)
    else
      return (CT_OUTSIDE, NULL)
  p ← SPLITTER(node)
  ℓ ← L-CHILD(node), r ← R-CHILD(node)
  switch CLASSIFY-LS(start, end, p)
    case BACK:
      return FIND-TRANS-S(start, end, r)
    case COPLANAR:
      tr1 ← FIND-TRANS-S(start, end, ℓ)
      tr2 ← FIND-TRANS-S(start, end, r)
      (cl1,pt1) ← tr1
      (cl2,pt2) ← tr2
      if cl1 = cl2 then
        switch cl1
          case CT_INSIDE or CT_OUTSIDE:
            return tr1
          case CT_TRANSITION:
            d1 ← DISTANCE²(start, pt1)
            d2 ← DISTANCE²(start, pt2)
            if d1 < d2 then return tr1
            else return tr2
      else if cl1 = CT_TRANSITION then
        return tr1
      else if cl2 = CT_TRANSITION then
        return tr2
      else return (CT_TRANSITION, start)
    case FRONT:
      return FIND-TRANS-S(start, end, ℓ)
    case STRADDLE:
      mid ← FIND-INTERSECT(start, end, p)
      cp ← CLASSIFY-P(start, p)
      if cp = FRONT then
        near ← L-CHILD(node)
        far ← R-CHILD(node)
      else
        near ← R-CHILD(node)
        far ← L-CHILD(node)
      tr1 ← FIND-TRANS-S(start, mid, near)
      (cl1,pt1) ← tr1
      if pt1 != NULL then return tr1
      tr2 ← FIND-TRANS-S(mid, end, far)
      (cl2,pt2) ← tr2
      if pt2 != NULL then return tr2
      if cl1 = cl2 then return tr1
      else return (CT_TRANSITION, mid)
```

Figure A.12: The effects of a light breeze on our hair simulation: the hair is being blown to the left by a wind coming from the right.

Figure A.13: Hair being applied to two separate regions of a horse model to form its mane and tail.

Figure A.14: A front view of the same horse model.

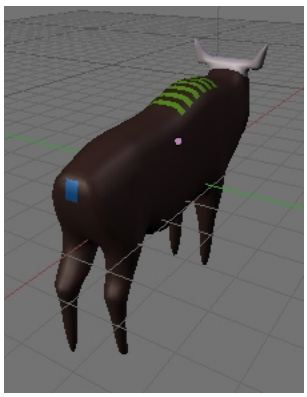


Figure A.15: Using Blender to apply different hair regions to a model of a bearded gnu. Each hair region is marked for hair distribution using a ‘special’ material (the green and blue mesh areas in the figure). Different materials encode different hair lengths, colours, etc. for later use.

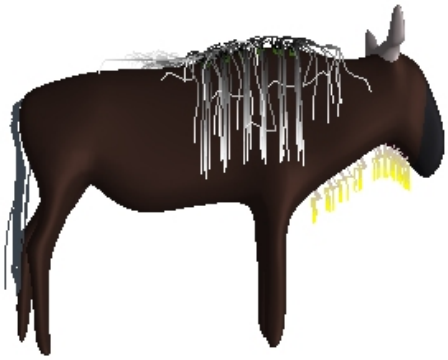


Figure A.16: The finished gnu, as rendered in our hair simulator. Note the three separate hair regions and their varying properties.

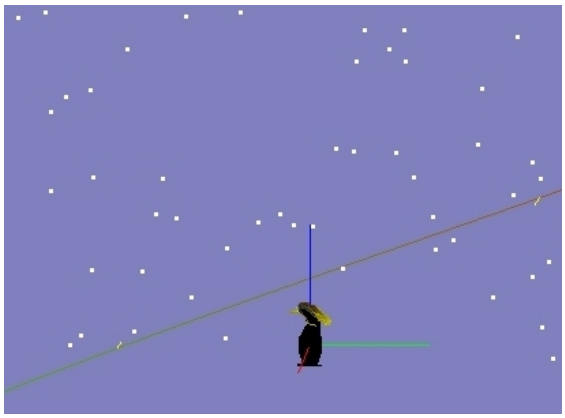


Figure A.17: Following a curved path around a long-haired penguin in falling snow. The hair can be seen falling over the penguin’s head and being blown sideways by the wind.

Appendix B

Watersheds and Waterfalls (Part 1)

Note: The text of this section is a verbatim extract from [18].

B.1 The Landscape Analogy

Anyone who's ever tried writing a terrain renderer will be familiar with the concept of using a heightmap to represent a landscape. Essentially, you have a 2D array of values, which can be viewed as an evenly-spaced finite grid located in the (x, y) plane. Each value represents the z height of the landscape at that point. (More formally, we could say that we have a rectangular domain $\Omega \subset \mathbb{Z}^2$ and a function $f : \Omega \to \mathbb{Z}$ which gives the height of the landscape for every point in Ω.)

The insight behind the watershed transform is that a grey-scale image is nothing but such a 2D array of values, so it can be viewed as a landscape, where the heights are given by the grey levels in the image (see Figure B.1).

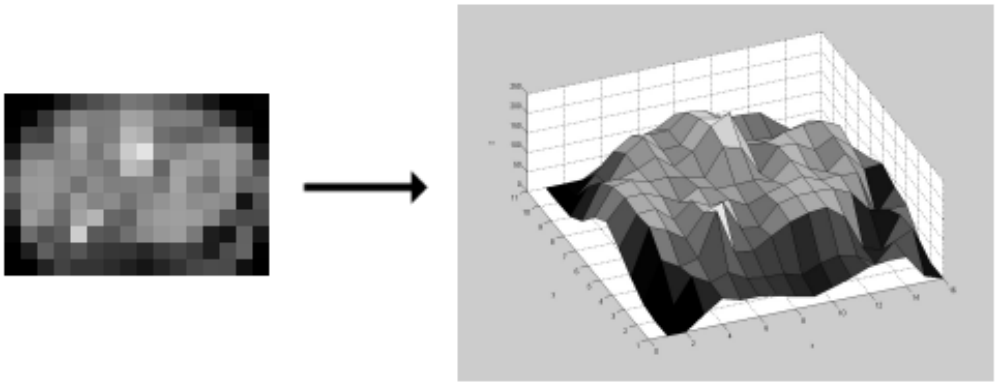


Figure B.1: Viewing an image as a landscape

We now define a few terms. A pixel $p \in \Omega$ has height $f(p)$ and *neighbour set* N(p), according to some implementation-specific definition of neighbourhood (usually pixels are considered to be either 4-connected or 8-connected: see Figure B.2).

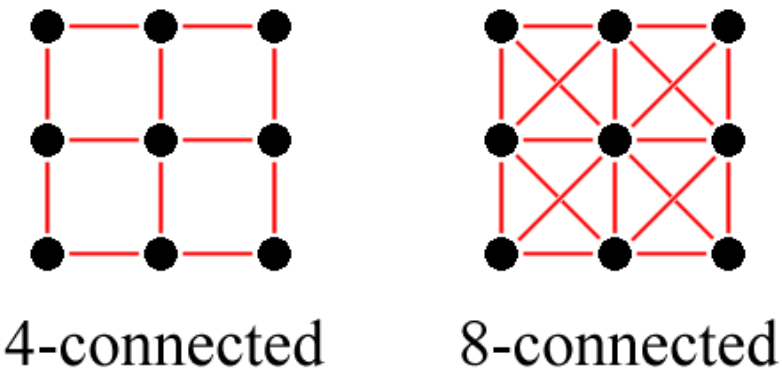
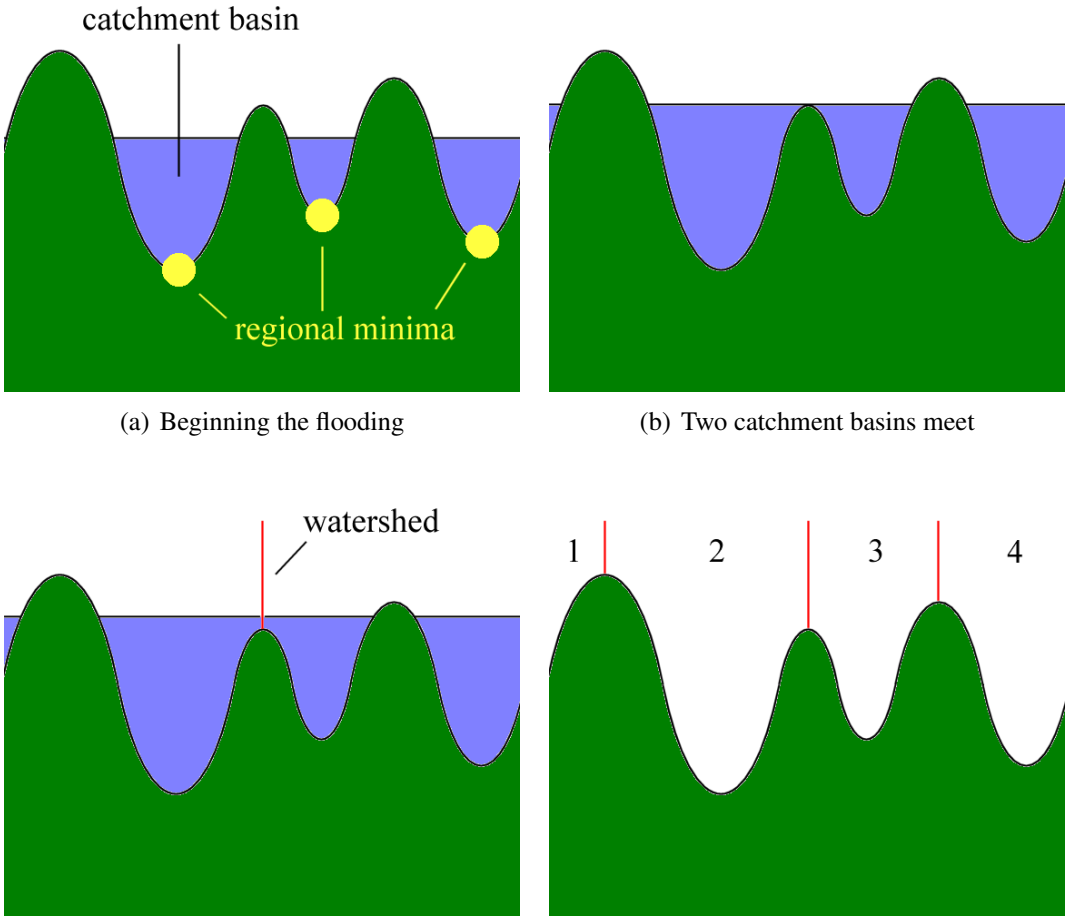


Figure B.2: 4-connected pixels are only connected to their horizontal and vertical neighbours, whereas 8-connected ones are connected to their diagonal neighbours as well

A *singular minimum* of the image is a point whose neighbours are all strictly higher than it. (More formally, p is a singular minimum if for all $p' \in N(p)$, $f(p') > f(p)$.) A *plateau* of the image is a maximal set of (two or more) connected pixels of equal altitude. A *minimal plateau* is a plateau from which it is impossible to descend, and a *non-minimal plateau* is the opposite. Together, the *singular minima* and *minimal plateaux* of the image form the *regional minima* of the image.

B.2 Flooding the Image Landscape

Imagine poking holes in the landscape at each of its regional minima, then lowering the landscape slowly into a lake. As the water begins to rise, pools of water will gradually form at each of the minima (see Figure B.3(a)). For reasons which will become obvious later, we'll call each pool of water the *catchment basin* of its associated minimum. If the water keeps rising, eventually some of the catchment basins will meet (see Figure B.3(b)): at this point, we imagine constructing a dam, or *watershed*, to keep them apart (see Figure B.3(c)) and continue the flooding. When the landscape has been fully flooded, the dams we've created will separate the different regional minima from each other at the points where their catchment basins would have met, thus segmenting the image into a number of regions, each associated with a different regional minimum (see Figure B.3(d)).



(a) Beginning the flooding

(b) Two catchment basins meet

(c) Building a dam at the join point

(d) The final division into regions

Figure B.3: The watershed construction process

This all sounds fine in theory, but there are a number of problems to be overcome:

1. There's no reason to suppose that the things we want to segment (e.g. organs in a medical image) will have low grey levels and thus be in a 'valley'. If they are on top of a 'hill', we're apparently in trouble.

2. The idea of the algorithm is one thing, but implementing it from this definition is far from straightforward.

3. Images can have large numbers of regional minima, especially in the presence of noise. Most of them are irrelevant, but the end result is that the image will end up being greatly oversegmented.

B.3 Using the Gradient Image

The first problem is definitely image-specific. However, for medical images, we're helped greatly by the fact that the organs we're segmenting (e.g. the kidneys and liver) tend to be relatively homogeneous, i.e. the grey levels are relatively similar throughout the organ. This implies that the image landscape is relatively flat over each organ, or in other words that the gradient is small there. By contrast, the gradient at the edges of organs will, we hope, be quite large. By using the gradient of the original image, then, instead of the image itself, we engineer a situation where the things we're trying to segment are (by and large) in the valleys of the image, and are (more or less) surrounded by hills (see Figure B.4).

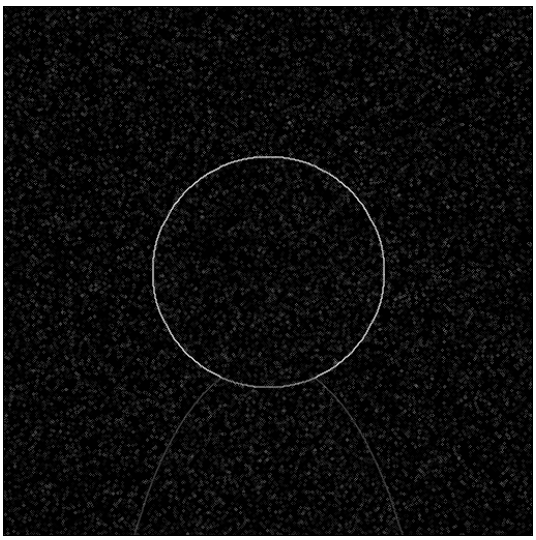


Figure B.4: In the gradient image, homogeneous features of interest end up in the valleys

B.4 Rainfall Simulation

Implementing the algorithm using a flooding method is only one way of approaching the problem. It's certainly possible to implement it that way (e.g. [54]), but it's sometimes helpful to think about things from a different perspective. Instead of thinking of the watershed process as one of flooding, we can now imagine rain falling on each point of the landscape from above.

Water is notorious for taking the path of least resistance, and in this case that means running downhill to a regional minimum via a path of *steepest descent* (i.e. a path where at each stage we choose to move to a lowest neighbour of the current point). This gives us an idea for an alternative approach to the watershed transform: we can think of the catchment basin of a regional minimum as

including all points whose unique path of steepest descent leads to the minimum in question. Where a point has more than one path of steepest descent, it can be allocated to any of the resulting minima according to programmer preference (see Figure B.5).

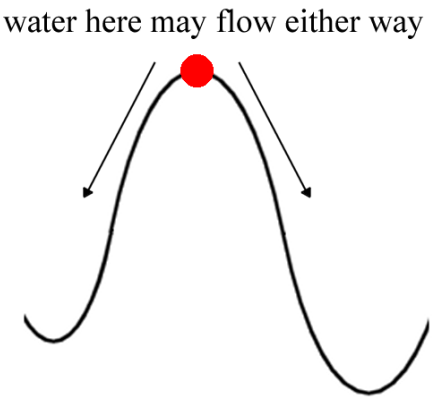


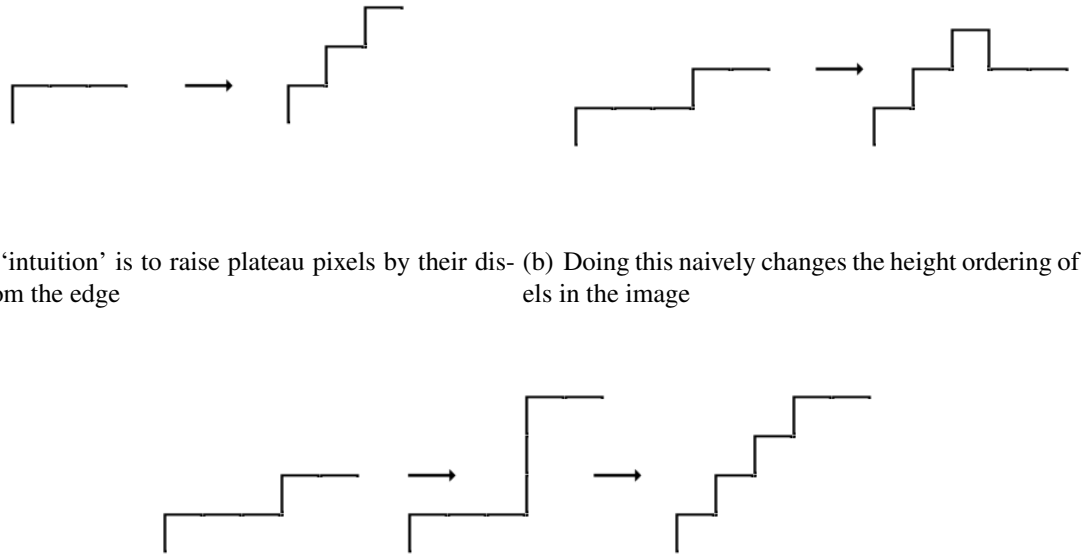
Figure B.5: The flow direction at a point is ambiguous if it has more than one path of steepest descent

B.5 Dealing with Non-Minimal Plateaux

A problem occurs when we think about which way water should run off a non-minimal plateau. There are various different approaches to dealing with this e.g. [6, 47, 63]; for the purposes of this article, we're going to use the approach in [42] and transform the image to remove all non-minimal plateaux at the outset, thus making it what is called *lower-complete*.

In essence, the idea is to raise all plateau pixels up by their distance from the plateau edge (see Figure B.6(a)). Of course, doing this naively doesn't work, since we might end up changing the ordering of the plateau pixels with respect to the other pixels in the image (see Figure B.6(b)). The solution, then, is to find the maximum amount by which we're going to raise a plateau pixel and multiply the base image by that before raising any pixels on the plateau: this has the effect of 'spreading the landscape out' to accommodate the new altitudes in the middle (see Figure B.6(c)).

Implementing this is relatively straightforward using a queue (see Listing 2 on Page 70). The basic idea is to add all pixels with a lower neighbour to the queue at the start, then gradually flood out from them a level at a time (essentially a breadth-first search), incrementing the distance counter after each level.



(a) The 'intuition' is to raise plateau pixels by their distance from the edge

(b) Doing this naively changes the height ordering of pixels in the image

(c) This can be fixed by 'spreading the landscape out' prior to raising the pixels

Figure B.6: A solution to the non-minimal plateau problem

B.6 Fletching

Having constructed a lower-complete image (see Figure B.7(b)), the rest is all downhill (excuse the pun). Our next step is to construct an arrow on each node (see Figure B.7(c)). In the case of a regional minimum, the arrow is a self-loop back to the node itself (for a minimal plateau, one of the nodes is chosen as a canonical element of the plateau and all the other nodes point to it); for all other nodes, the arrow points to a lowest neighbouring node (i.e. it points in the direction of a path of steepest descent). We also take the opportunity to numerically label all the canonical elements during this phase of the process.

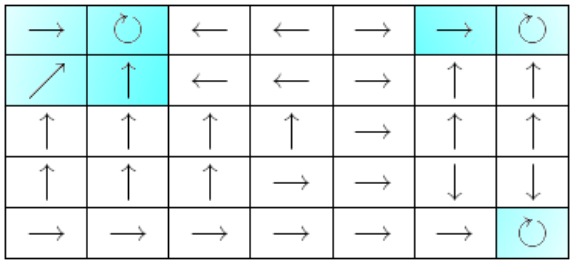
The implementation (see Listing 3 on Page 71) uses an interesting disjoint-set forest data structure which I'll talk about in more detail next time. The idea is to combine all the minimum points into their respective regional minima using this data structure, and make all the other non-minimal points point to one of their lowest neighbours.

2	2	3	4	4	1	1
2	2	3	4	4	2	2
4	4	5	5	5	4	6
6	6	5	5	5	2	6
6	6	5	5	5	2	1

(a) The original image

0	0	9	12	12	0	0
0	0	9	12	12	6	6
12	12	15	15	15	12	18
18	18	16	16	15	7	18
19	18	17	16	15	6	0

(b) The lower-complete image



(c) The arrows on each node

0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	2	2	2	2
2	2	2	2	2	2	2

(d) The final labelling

Figure B.7: Stages of the algorithm

B.7 Labelling the Image

The final step of the basic watershed algorithm is to label the pixels (see Listing 4 on Page 72). This involves following the arrows for each pixel to find which minimum it's associated with, and giving it the same label as that minimum. To speed things up, we use *path compression* when following a path to a minimum (i.e. we make all the arrows on the path point to the minimum once we've found it). Interestingly, this bears many similarities to the implementation of the disjoint-set data structure I just mentioned: we'll see more of this next time.

The result (see Figure B.7(d)) is, in general, an oversegmented image on which further processing is then required.

B.8 Conclusion

At this stage, we can be tolerably pleased with our efforts. We've managed to segment the image into a number of regions, each associated with a regional minimum of the image, but we haven't yet got what we need. In particular, our image is greatly over-segmented, because most of the regional minima aren't 'relevant': they're not associated with the objects of interest in the image. A general method to solving this problem involves trying to merge some of the regions together to reduce the overall number of regions in our image and obtain a better segmentation. One algorithm which takes this approach is the *waterfall algorithm* described in [40], which I'll talk about next time. In and of itself, this still won't give us what we need for medical images, but it will take us a little closer to where we need to be. It also produces reasonable results on some non-medical images (e.g. the ones used in the paper). To get acceptable results for medical images, we have to make use of anatomical knowledge to process the results of application-independent algorithms like the waterfall, but that's something that is very much still a work in progress!

Listing 2 Building a lower-complete function

```
Build-Lower-Complete(Function<Ω,Z> image)

Function<Ω,Z> lc;
Queue<PixelCoords> queue;
// A marker indicating when we need to increase the distance value.
PixelCoords marker(-1,-1);

// Initialise the queue with pixels that have a lower neighbour.
foreach(PixelCoords p ∈ Ω)
    lc(p) = 0;
    foreach(PixelCoords neighbour ∈ N(p))
        if(image(neighbour) < image(p))
            queue.push(p);
            // To prevent it being queued twice.
            lc(p) = -1;
            break;

// Compute a function which indirectly indicates the amount by which
// we need to raise the plateau pixels (see the referenced paper for
// more details).
int dist = 1;
queue.push(marker);
while(!queue.empty())
    p = queue.pop();
    if(p == marker)
        if(!queue.empty())
            queue.push(marker);
            ++dist;
    else
        lc(p) = dist;
        foreach(PixelCoords neighbour ∈ N(p))
            // If the neighbouring pixel is at the
            // same altitude and has not yet been
            // processed.
            if(image(neighbour) == image(p) && lc(neighbour) == 0)
                queue.push(neighbour);
                // To prevent it being queued twice.
                lc(neighbour) = -1;

// Compute the final lower-complete function. Note that at this point,
// dist holds the amount by which we want to multiply the base image.
foreach(PixelCoords p ∈ Ω)
    if(lc(p) != 0)
        lc(p) = dist * image(p) + lc(p) - 1;

return lc;
```

Listing 3 Constructing arrows on the nodes

```
Construct-Arrows(Function<Ω,Z> lc)

Function<Ω,PixelCoords> arrows;
Function<Ω,PixelCoords> labels;

// Add all the minimum points to a disjoint set forest.
int labelCount = 0;
DisjointSetForest<PixelCoords> minima;
foreach(PixelCoords p ∈ Ω)
    if(lc(p) == 0)
        labels(p) = labelCount++;
        minima.add_node(p);

foreach(PixelCoords p ∈ Ω)
    if(lc(p) == 0)
        // Union any neighbouring minimum points into the same regional minimum.
        foreach(PixelCoords neighbour ∈ N(p))
            if(lc(neighbour) == 0)
                minima.union_nodes(labels(p), labels(neighbour));
    else
        // Find a lowest neighbour and make this point's arrow point to it.
        PixelCoords lowestNeighbour(-1,-1);
        int lowestNeighbourValue = INT_MAX;
        foreach(PixelCoords neighbour ∈ N(p))
            if(lc(neighbour) < lowestNeighbourValue)
                lowestNeighbour = neighbour;
                lowestNeighbourValue = lc(neighbour);
        // There will always be a lowest neighbour here since the function's
        // lower-complete.
        arrows(p) = lowestNeighbour;

// Assign new labels to the canonical points of the regional minima and make
// the arrows of the non-canonical points point to them.
labelCount = 1;
foreach(PixelCoords p ∈ Ω)
    if(lc(p) != 0) continue;
    int root = minima.find_set(labels(p));
    if(root == labels(p))
        // This is a canonical point.
        arrows(p) = p;
        labels(p) = labelCount++;
    else
        arrows(p) = minima.value_of(root);

return (arrows, labels);
```

Listing 4 Labelling all the pixels by following the arrow chains

```
Resolve-All(Function<Ω,PixelCoords> arrows, Function<Ω,Z> labels)

foreach(PixelCoords p ∈ Ω)
    Resolve-Pixel(p, arrows, labels);

Resolve-Pixel(PixelCoords p, Function<Ω,PixelCoords> arrows,
              Function<Ω,Z> labels)

PixelCoords parent = arrows(p);
if(parent != p)
    parent = Resolve-Pixel(parent);
    labels(p) = labels(parent);
return parent;
```

Appendix C

Watersheds and Waterfalls (Part 2)

Note: The text of this section is a verbatim extract from [18].

C.1 Introduction

In my last article [18], I described a way of segmenting images using the watershed transform and commented that the biggest problem with the results was one of *oversegmentation*: the image gets divided into too many regions because a region is generated for every regional minimum in the image, regardless of whether it's of any interest to us. The waterfall algorithm [40], the subject of this article, is a hierarchical approach which attempts to solve this problem. (Readers may wish to consult the original paper for a more detailed justification of some of the methods involved.)

C.2 Gaussian Blurring

Before we start looking at the algorithm in detail, though, it's worth observing that there are useful pre-processing steps we can take to reduce the initial number of regional minima before even applying the watershed transform. In particular, it's well worth our time to apply a Gaussian blur to the original image before taking its gradient.

Gaussian blurring is essentially a form of weighted pixel averaging based on a discrete approximation to the 2D version of the normal distribution. The 1D Gaussian from statistics is a familiar concept:

$$g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-x^2}{2\sigma^2}\right)$$

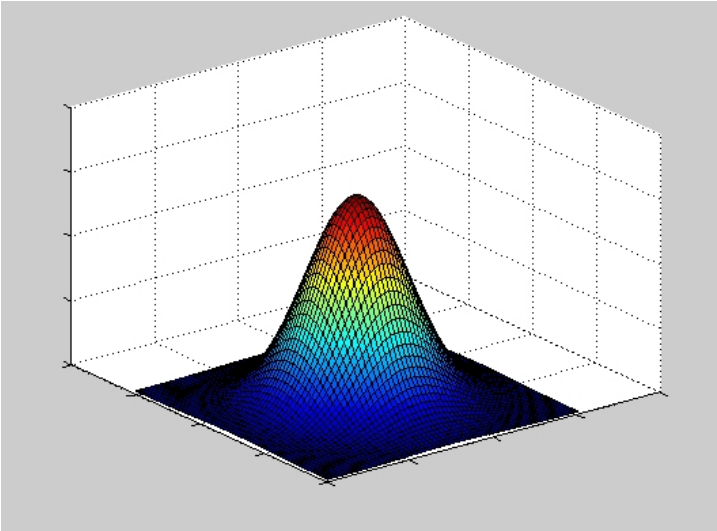


Figure C.1: The 2D Gaussian is a bell-shaped surface

Its 2D version can be obtained by multiplying a 1D Gaussian in the x direction with one in the y direction:

$$G_\sigma(x,y) = g_\sigma(x) \times g_\sigma(y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

In 1D its graph is the familiar bell-shaped curve; in 2D we get a bell-shaped surface (see Figure C.1).

To use this for image blurring, we form a symmetric mask (see Figure C.2) from the values of $G_\sigma(x,y)$ at discrete points in a grid centred at the origin (e.g. for a 3x3 mask, we calculate values at $(-1,-1),(0,-1),(1,-1),\ldots,(0,0),\ldots,(1,1)$). We then normalize the mask by dividing by the sum of all the values in it (this is done to ensure that regions of uniform intensity in the image will be unaffected by smoothing). This procedure can be used to generate masks of other sizes as well.

The actual blurring is done by what is known as *convolving* the image with the mask. This basically means overlaying the mask on each pixel of the image in turn, multiplying the value of each pixel in the mask by the value of the pixel beneath it, summing the results and using the value thus obtained as the value of the centre pixel in the blurred image. For the 3x3 mask with $\sigma = 1$, this means that if $I(x,y)$ is the source image, $M(x,y)$ is the mask and $I'(x,y)$ is the blurred image, then:

$$\begin{aligned} I'(x,y) \;=\; & 0.0751 \times (I(x-1,y-1) + I(x+1,y-1) + I(x-1,y+1) + I(x+1,y+1)) \;+ \\ & 0.1238 \times (I(x,y-1) + I(x,y+1) + I(x-1,y) + I(x+1,y)) \;+ \\ & 0.2042 \times I(x,y) \end{aligned}$$

0.0585	0.0965	0.0585
0.0965	0.1592	0.0965
0.0585	0.0965	0.0585

$\longrightarrow$

0.0751	0.1238	0.0751
0.1238	0.2042	0.1238
0.0751	0.1238	0.0751

Figure C.2: As an example, we'll calculate a 3x3 mask for the 2D Gaussian $G_1(x, y)$ (i.e. the Gaussian with standard deviation $\sigma = 1$). First we calculate the values of $G_1(x, y)$ at the grid points (i.e. we calculate $G_1(-1, -1), \ldots, G_1(1, 1)$) to give us the unnormalized mask (left); then, we normalize it by dividing through by the sum of all the values in the mask to give the final result (right).

C.3 Introducing the Waterfall

Having talked about pre-processing, we can now turn our attentions to the actual waterfall algorithm. The basic idea is to take the result of the watershed transform on the gradient of the original image and use it to produce a sequence of images by merging some adjacent regions (see Figure C.3).

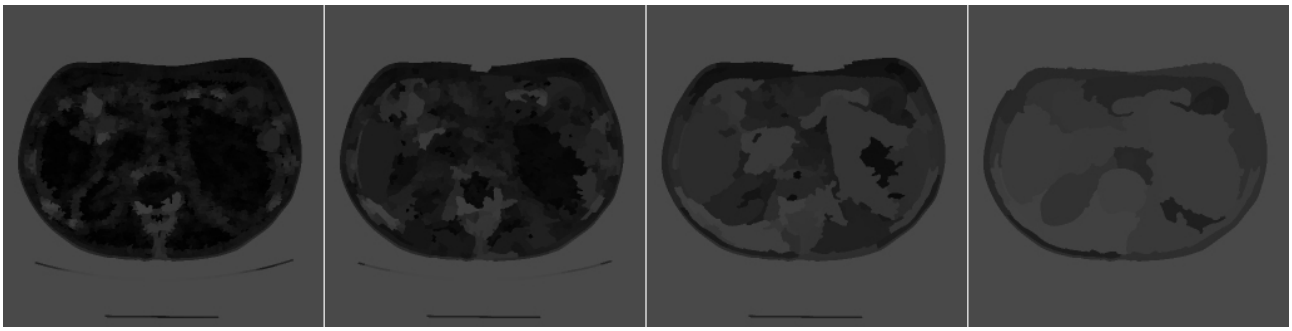


Figure C.3: The waterfall algorithm produces a hierarchical sequence of segmentations, starting from the original watershed result (far left). The final image will eventually be a single region (not shown).

The algorithm described in [40] works on the *region adjacency graph* (RAG) of the watershed result. This is a graph with one vertex for each region in the watershed, and weighted edges joining adjacent regions (see Figure C.4). As we will see, the weights on the edges essentially determine the order of region merging, and we have a number of different options when calculating them. For now, we'll assume that we already have a suitably weighted graph, and focus on how to use it to iterate from one stage in the waterfall sequence to the next.

The basic idea of a waterfall iteration involves doing something very like a watershed algorithm on the RAG. First of all, we have to find the regional minima of the graph, which in this case means its *regional minimum edges* (we'll define what we mean by this shortly). We then mark each such edge with a different label and carefully propagate the labels to the rest of the edges of the graph (this is an implementation of the watershed-from-markers algorithm, where the regional minimum edges are the markers). This induces a new labelling of the various regions, resulting in some of the adjacent

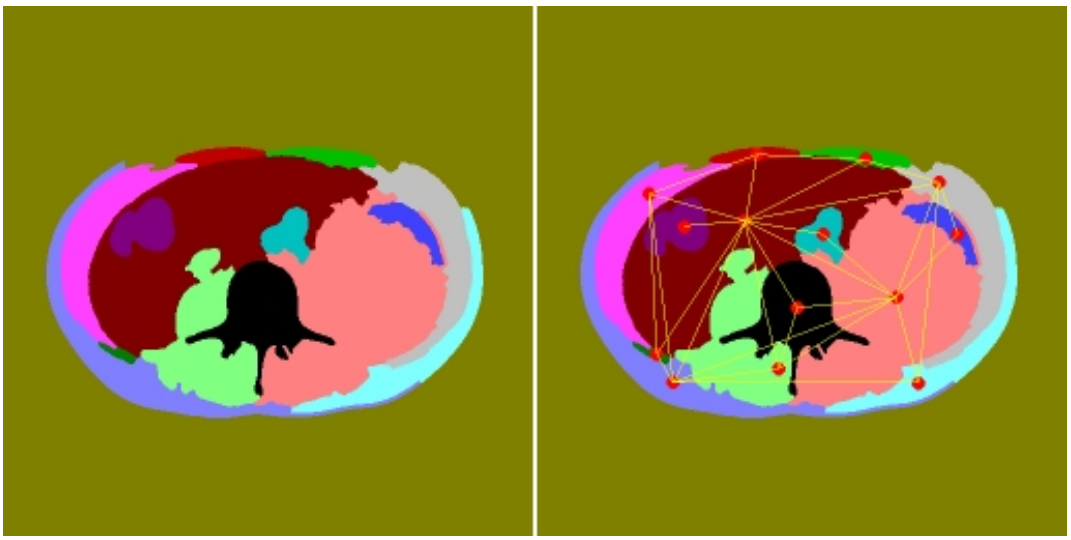


Figure C.4: A set of regions (left) and their region adjacency graph (right) - note that edges to the surrounding region are not shown to make things clearer

regions being merged.

In practice, we don't run the algorithm on the RAG itself; for reasons that are fully explained in the referenced paper, the *minimum spanning tree* (MST) of the graph contains sufficient information that we can simply run the algorithm on that, with a corresponding gain in efficiency. To briefly review MSTs, they can be defined as follows. Given a graph $G = (V, E, w)$ with vertex set V, edge set E and weight function $w : E \to \mathbb{Z}^+$, the set of spanning trees $ST(G)$ of G is the set of subgraphs of G which are both trees (i.e. they're acyclic) and which span G (i.e. they contain every vertex in V): see Figure C.5 for an example.

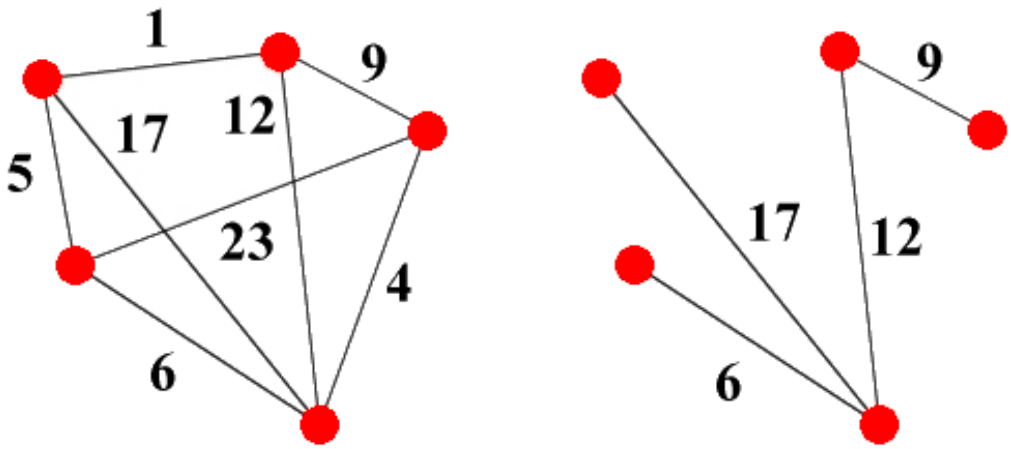


Figure C.5: A graph and one possible spanning tree for it

A *minimum* spanning tree is then simply one with a minimum total cost, i.e. a spanning tree $T = (V, E', w) \in ST(G)$ such that

$$\forall (V, E'', w) \in ST(G) \cdot \sum_{e' \in E'} w(e') \leq \sum_{e'' \in E''} w(e'')$$

Constructing a minimum spanning tree can be done straightforwardly using Kruskal's algorithm. This involves sorting the edges in the graph into ascending order by weight, then adding the edges in ascending order to the minimum spanning tree provided they wouldn't create a cycle and invalidate the tree.

C.4 Data Structures

To implement the waterfall efficiently, we're going to have to learn a bit about data structures. One of the key things we need to know about is Tarjan's data structure for disjoint set forests. This structure, designed for maintaining a collection of (mutually) disjoint sets that change over time, is widely useful and not merely restricted to our current purposes. Indeed it's the sort of thing that crops up in Computer Science degree courses [74]! This structure gets used during the fletching stage of the watershed algoritm and as part of Kruskal's algorithm, and that's before we've even mentioned its usage in maintaining the regions for the actual waterfall algorithm itself.

The idea, then, is to represent each disjoint set as a rooted tree. Finding which set an element is in is as simple as walking up the tree to the root. Unioning two sets involves finding the roots of two separate trees, and making one tree root a child of the other. For efficiency reasons, it makes sense to keep the paths to the roots of the trees as small as possible. Two tricks used to accomplish this are *union-by-rank* and *path compression*. Without dwelling on the details, the first of these tries to ensure that we're making the root of the smaller tree a child of that of the larger one, and the second changes all the parent pointers on a path to the root to point directly to the root when a 'find root' call is made: this ensures that subsequent calls on any element on the path will be constant time. The code to implement all this is shown in Listing 5 below.

Listing 5 Tarjan's Disjoint Set Forest

```
MAKE-SET(x)
  parent[x] ← x
  rank[x] ← 0

FIND-SET(x)
  if x ≠ parent[x] then
    parent[x] ← FIND-SET(parent[x])
  return parent[x]

LINK(x,y)
  if rank[x] > rank[y] then
    parent[y] ← x
  else
    parent[x] ← y
    if rank[x] = rank[y] then
      rank[y] = rank[y] + 1

UNION(x,y)
  LINK(FIND-SET(x), FIND-SET(y))
```

In the waterfall algorithm, we use such a disjoint set forest to store which regions are connected to each other. Initially, we have a tree for each region in the watershed result; as we merge regions (see Edge Elision below), we then union their respective trees. This makes region merging a very fast process, since we don't have to update the region indices for all the individual pixels in the regions. Instead, each pixel maintains the label it was originally given by the watershed transform: this can then be used to look up the correct region value in the disjoint set forest associated with any given level of the waterfall. The space savings are also noticeable: instead of storing a full image of labels for each waterfall iteration, we need only store the results of the watershed and a disjoint set forest for each level of the hierarchy.

The other data structures we'll use are for maintaining edges. The layout is as shown in Figure C.6. We maintain an array (in practice, a std::vector) which stores all the edges we'll be referring to (these can either be all the edges in the RAG, or all the edges in the initial MST if we want to be particularly space-efficient). The MST is represented as a list of edge pointers sorted in ascending order of edge weight. Finally, we store an edge adjacency table, which stores lists of pointers to edges which are adjacent to each of the various regions.

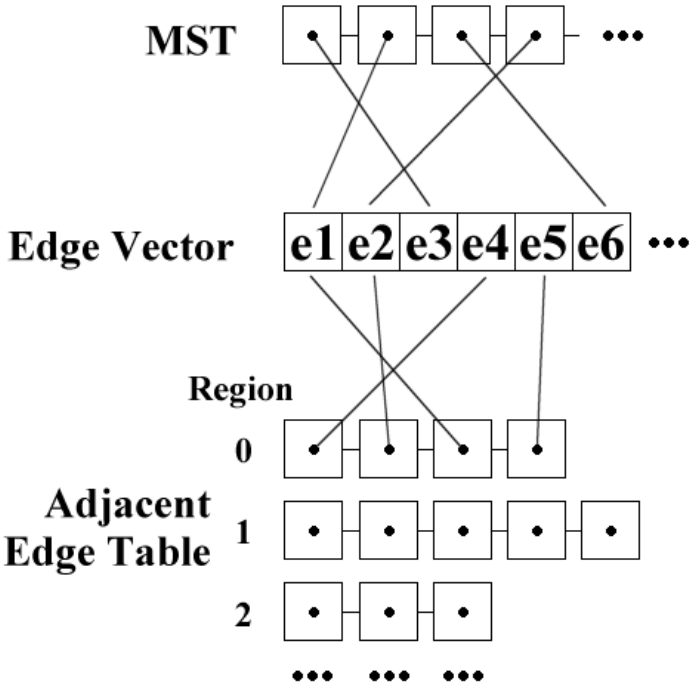


Figure C.6: Data structures used for the waterfall algorithm: some of the pointers aren't shown for reasons of clarity.

C.5 Step 1: Finding the Regional Minimum Edges

A *regional minimum edge* (RME) of a graph G is a connected subgraph of G whose own edges have equal weight and whose adjacent edges in G have strictly higher weights (see Figure C.7).

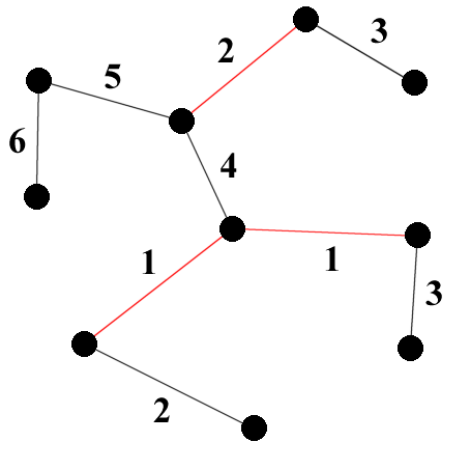


Figure C.7: An example graph and its RMEs (marked in red). Note that the two edges with a weight of 1 are part of the same RME.

To find all the regional edges in the MST, we run through all the edges in the MST and flood outwards from each one to determine (a) whether it's part of an RME and (b) the extent of the RME if so (see Listing 6). We do this by maintaining two things: an *equal edges* list, which holds all the edges that may form part of the current hypothetical RME, and an *adjacent edges* queue, which holds all unprocessed edges which are adjacent to one of the aforementioned equal edges. We process all adjacent edges one at a time. If we see a lower edge, this isn't an RME. If we see an equal edge, we add it to the list of edges which may be in the RME and add its adjacent edges to the queue. If we see a higher edge, we ignore it. If we empty the adjacent edges queue without seeing a lower edge, we've found an RME, so we add it to the list, mark all the edges in it as being part of an existing minimum (to avoid later duplication) and continue from the top with the next MST edge.

C.6 Edge Elision

In my implementation of the waterfall, no actual relabelling of the regions is done. Instead, the same effect is achieved by merging regions via the mechanism of *eliding* (i.e. removing) edges in the RAG. The process of edge elision is slightly intricate because all the relevant data structures need to be updated. We need to union the disjoint set forest trees associated with the regions at either end of the edge, we need to splice the edge adjacency lists together (thus adding the list of all the edges adjacent to one of the regions to that of the other) and we need to do various bits of additional housekeeping (see Listing 7). One important step is to mark the edge as elided, so that it can be removed from the MST in Step 4 of the algorithm (see below).

C.7 Step 2: Eliding the RMEs

Having found the RMEs in Step 1 (above), the next part of the actual algorithm is to elide them. This is done as an alternative to assigning them all the same label (as per the original waterfall description).

C.8 Step 3: Marker Propagation

Once we've 'labelled' the marker regions by eliding all the RMEs connecting them, the next step is to propagate the markers to the rest of the MST using a flooding process, at each stage processing an

adjacent edge with lowest cost. The ideal data structure for this is a priority queue. The algorithm (see Listing 8) works as follows: we initialise the queue with any edges which are adjacent to RMEs and mark the regions joined by the RMEs as already processed. We then repeatedly pop the lowest cost edge from the queue, merging the regions it connects if one of them is unmarked, and ignoring it otherwise. If an edge is elided, its own adjacent edges are also added to the queue.

C.9 Step 4: Rebuilding the MST

The final step of the algorithm is to rebuild the MST so that it's ready for the next waterfall iteration. This turns out to be an almost trivial process, since we just have to remove any elided edges from the tree. Since we carefully marked them all with a value of -1 when they were elided, all we have to do is run through the list representing the MST and remove any edges whose value is -1 (in C++, this can be handled extremely simply using a remove_if call).

C.10 Edge Valuations

One issue we haven't yet touched on is how to generate suitable weights for the edges of the region adjacency graph. Since these weights determine the order of region merging, they have a substantial effect on the output of the whole algorithm, so it's important to choose them carefully.

There are a number of different options available. The simplest approach, known as *lowest pass*, valuates each edge with the height of the lowest pass point on the border between the regions it joins. One advantage of this method is that it's easy to calculate: you just run through all the pixels in the watershed result, find any border pixels (pixels which have at least one neighbour with a different label) and update the lowest pass between any two regions as necessary.

Lowest pass isn't always the best method to use, however, and several other sensible valuations have been proposed. These generally focus on the idea of *dynamics*, which involves thinking about either the height (contrast dynamics), the surface area (area dynamics) or the volume (volume dynamics) of the water in the catchment basins of the adjacent regions at the point when they would meet during the flooding process (i.e. the point where a watershed is built to keep them apart). These all produce different results and some experimentation is needed to see which may be the most appropriate in a given situation. Another interesting valuation can be found in [9], where the authors use some

knowledge about the human perception of shapes to define a dynamic which (on their test images at any rate) produces more visually-pleasing results than (in particular) the volume dynamic, with which they contrast it.

My own work is currently using lowest pass, but I plan to experiment further with the other valuations in due course.

C.11 Conclusion

From my own experiments with the waterfall algorithm, I can attest to the fact that the results of waterfall segmentation seem to be much more useful than the original watershed result. It's not that any particular image in the waterfall sequence gives us the ideal segmentation: that would be far too easy. What the waterfall does give us is a lot of connected regions (in the various different images in the sequence) to work with further. With these results, we can go on to use region analysis and classification strategies to identify which regions in the various waterfall iterations correspond to features of interest in our original image. Waterfall thus takes us one step closer to our original goal of automatic segmentation. How to do the actual region analysis and classification is still a research problem, but one I hope to continue working on in the near future.

Listing 6 The flooding algorithm for finding RMEs

```
Vector<EdgeList> rmeArray;

foreach(Edge startEdge ∈ mst)
    // The edge is already part of a regional minimum edge. If we processed it,
    // we'd end up duplicating that minimum edge, so skip over it.
    if(startEdge->minimum) continue;

    EdgeList equalEdges;      // maintain a list of equally-valued edges which
                              // may form part of the same RME
    EdgeQueue adjacentEdges;
    EdgeQueue pendingEdges;   // maintain a queue of pending edges so they can
                              // be unmarked again later

    // Mark the initial edge to ensure it isn't wrongly identified as being
    // adjacent to itself.
    startEdge->pending = true;
    pendingEdges.push(startEdge);

    add_adjacent_edges(startEdge, adjacentEdges, pendingEdges);

    bool isMinimum = true;
    while(!adjacentEdges.empty())
        Edge adjacent = adjacentEdges.pop();

        // Compare its value to the start edge.
        if(adjacent->m_value < startEdge->m_value)
            // If its value is less than the start edge, then the start edge
            // is not a minimum.
            isMinimum = false;
            break;
        else if(adjacent->m_value == startEdge->m_value)
            equalEdges.push_back(adjacent);
            add_adjacent_edges(adjacent, adjacentEdges, pendingEdges);

    // Unmark all the pending edges.
    while(!pendingEdges.empty())
        Edge e = pendingEdges.pop();
        e->pending = false;

    if(isMinimum)
        equalEdges.push_front(startEdge); // add the start edge to the list now
                                          // that we know we've found a minimum
        for(Edge e ∈ equalEdges)
            e->minimum = true;
        rmeArray.push_back(equalEdges);
```

Listing 7 Edge elision

ELIDE-EDGE(e)

```
unsigned int u = e->u;
unsigned int v = e->v;

unsigned int setU = forest.find_set(u);
unsigned int setV = forest.find_set(v);

forest.union_nodes(u, v);

// Mark the edge as elided for when we come to
// later rebuild the MST.
e->value = -1;

// forest.find_set(u) == forest.find_set(v)
unsigned int parent = forest.find_set(u);

// Add all the edges adjacent to the the child regions to the parent region in
// the adjacency table.
EdgeList parentList = adjacencyTable[parent];
if(setU != parent) parentList.splice(parentList.end(), adjacencyTable[setU]);
if(setV != parent) parentList.splice(parentList.end(), adjacencyTable[setV]);

// Remove the elided edge from the parent list.
parentList.remove(e);

// Update the region values.
Value parentValue = forest.value_of(parent);
Value uValue = forest.value_of(setU);
Value vValue = forest.value_of(setV);
if(uValue > parentValue) parentValue = uValue;
if(vValue > parentValue) parentValue = vValue;
```

Listing 8 Marker propagation

```
// Flags indicating whether a region has been marked or not.
Vector<char> markedRegion(forest.node_count());

PriorityQueue<Edge> pq;
EdgeQueue pendingEdges;

// Add all the edges adjacent to regional minimum edges to the priority queue.
foreach(EdgeList rme ∈ rmeArray)
    foreach(Edge e ∈ rme)
        add_adjacent_edges(e, pq, pendingEdges);
        int setU = forest.find_set(e->u);
        int setV = forest.find_set(e->v);
        markedRegion[setU] = markedRegion[setV] = 1;

// Process each edge in ascending order of value, adding adjacent edges to the
// priority queue each time.
while(!pq.empty())
    Edge e = pq.pop();

    int setU = forest.find_set(e->u);
    int setV = forest.find_set(e->v);

    // If this region connects a marked region to an unmarked one, it needs
    // processing. Otherwise it should be ignored. Note that because of the
    // way the propagation works, any edge will either connect a marked
    // region to an unmarked one, or it will connect two marked regions.
    if(!markedRegion[setU] || !markedRegion[setV])
        ELIDE-EDGE(e);
        markedRegion[setU] = markedRegion[setV] = 1;
        add_adjacent_edges(e, pq, pendingEdges);

// Unmark all the pending edges.
while(!pendingEdges.empty())
    Edge e = pendingEdges.pop();
    e->pending = false;
```

Bibliography

[1] T Alarcón, H M Byrne, and P K Maini. Towards whole-organ modelling of tumour growth. *Progress in Biophysics and Molecular Biology*, 85:451–472, 2004.

[2] Ferran Algaba. Is tumor necrosis a predictor of survival in patients with renal cell carcinoma?, April 2006.

[3] H Ancin, T E Dufresne, G M Ridder, J N Turner, and B Roysam. An improved watershed algorithm for counting objects in noisy, anisotropic 3-D biological images. In *Proceedings of the International Conference on Image Processing*, volume 3, pages 172–175, October 1995.

[4] Jamal Atif, Céline Hudelot, Geoffroy Fouquier, Isabelle Bloch, and Elsa Angelini. From Generic Knowledge to Specific Reasoning for Medical Image Interpretation using Graph based Representations. In *IJCAI*, pages 224–229, 2007.

[5] Serge Beucher. Geodesic Reconstruction, Saddle Zones & Hierarchical Segmentation. *Image Analysis and Stereology*, 20:137–141, 2001.

[6] Andreas Bieniek and Alina Moga. An efficient watershed algorithm based on connected components. *Pattern Recognition*, 33:907–916, 2000.

[7] Blender 3D: Noob to Pro. Penguins from spheres. `http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Penguins_from_spheres`.

[8] G F Cahill and M M Melicow. Calcification of renal tumors and its relation to prognosis. *Journal of Urology*, 39:276–286, 1938.

[9] Joan Climent and Alberto Sanfeliu. Visually Significant Dynamics for Watershed Segmentation. In *Proceedings of the 18th International Conference on Pattern Recognition (ICPR '06)*, 2006.

[10] Laurent D Cohen. On Active Contour Models and Balloons. *Computer Vision, Graphics and Image Processing: Image Understanding*, 53(2):211–218, March 1991.

[11] Alexander Cong, Yi Liu, D Kumar, Wenxiang Cong, and Ge Wang. Geometrical Modeling Using Multiregional Marching Tetrahedra for Bioluminescence Tomography. In *Medical Imaging 2005: Visualization, Image-Guided Procedures, and Display (Proceedings of the SPIE)*, volume 5744, pages 756–763, 2005.

[12] Dubravko Ćosić and Sven Lončarić. Rule-Based Labeling of CT Head Image. In *Proceedings of the 6th Conference on Artificial Intelligence in Medicine Europe*, pages 453–456, 1997.

[13] J S Lam et al. Clinicopathologic and molecular correlations of necrosis in the primary tumor of patients with renal cell carcinoma. *Cancer*, 103:2517–2525, 2005.

[14] T R Fetter. Renal carcinoma. A study of ninety five cases with follow-up notes on thirty six. *JAMA*, 110:190–196, 1938.

[15] V Foria, T Surendra, and D N Poller. Prognostic relevance of extensive necrosis in renal cell carcinoma. *Journal of Clinical Pathology*, 58:39–43, 2005.

[16] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by A Priori Tree Structures. *Computer Graphics*, 14(3):124–133, June 1980.

[17] Stuart Golodetz. Functional Programming Using C++ Templates (Parts 1 and 2). *Overload*, 81/82, October/December 2007.

[18] Stuart Golodetz. Watersheds and Waterfalls (Parts 1 and 2). *Overload*, 83/84, February/April 2008.

[19] Stuart Golodetz, Irina Voiculescu, and Stephen Cameron. Real-Time Dynamics Simulation of Long Hair over Arbitrary Meshes. Rejected by Eurographics 2008.

[20] Stuart M. Golodetz. A 3D Map Editor. Undergraduate thesis, Oxford University Computing Laboratory, May 2006.

[21] Sunil Hadap and Nadia Magnenat-Thalmann. Modeling dynamic hair as a continuum. *Computer Graphics Forum*, 20(3):329–338, 2001.

[22] Celina Imielinska, Yinpeng Jin, Elsa Angelini, Dimitris Metaxas, Ting Chen, Jayaram K Udupa, and Ying Zhuge. *Hybrid Segmentation Methods*, chapter 12, pages 351–388. A K Peters, 2004.

[23] A C Jalba, M H F Wilkinson, and J B T M Roerdink. Automatic Image Segmentation using a Deformable Model based on Charged Particles. In *Proceedings of the International Conference on Image Analysis and Recognition*, volume 3211 of Lecture Notes in Computer Science, pages 1–8, 2004.

[24] Yvonne Jung, Alexander Rettig, Oliver Klar, and Timo Lehr. Realistic Real-Time Hair Simulation and Rendering. In E. Trucco and M. Chantler, editors, *Proceedings of the International Conference on Vision, Video and Graphics, Edinburgh (UK)*, pages 229–236, July 2005.

[25] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, January 1988.

[26] Matthew Kelly, Olivier Noterdaeme, and Michael Brady. Relating intra-tumor heterogeneity to morphology and its implications for assessing response to therapy. In *IEEE International Symposium on Biomedical Imaging*, 2007.

[27] Tae-Yong Kim and Ulrich Neumann. A thin shell volume for modeling human hair. In *CA '00: Proceedings of the Computer Animation*, page 104, Washington, DC, USA, 2000. IEEE Computer Society.

[28] Masaharu Kobashi and Linda G Shapiro. Knowledge-Based Organ Identification from CT Images. *Pattern Recognition*, 28(4):475–491, 1995.

[29] Cord Langner, Georg Hutterer, Thomas Chromecki, Sebastian Leibl, Peter Rehak, and Richard Zigeuner. Tumor Necrosis as Prognostic Indicator in Transitional Cell Carcinoma of the Upper Urinary Tract. *Journal of Urology*, 176:910–914, September 2006.

[30] C K Lee, F W Choy, and H C Lam. Real-time Thresholding using Histogram Concavity. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, volume 1, pages 500–503, 1992.

[31] Chien-Cheng Lee, Pau-Choo Chung, and Hong-Ming Tsai. Identifying Multiple Abdominal Organs From CT Image Series Using a Multimodule Contextual Neural Network and Spatial

Fuzzy Rules. *IEEE Transactions on Information Technology in Biomedicine*, 7(3), September 2003.

[32] Sang Eun Lee, Seok-Soo Byun, Jin Kyu Oh, Sang Chul Lee, In Ho Chang, Gheeyoung Choe, and Sung Kyu Hong. Significance of Macroscopic Tumor Necrosis as a Prognostic Indicator for Renal Cell Carcinoma. *Journal of Urology*, 176:1332–1338, October 2006.

[33] Sang Eun Lee, Sung Kyu Hong, Byung Kyu Han, Ji Hyung Yu, June Hyun Han, Seong Jin Jeong, Seok-Soo Byun, Yong Hyun Park, and Gheeyoung Choe. Prognostic Significance of Tumor Necrosis in Primary Transitional Cell Carcinoma of Upper Urinary Tract. *Japanese Journal of Clinical Oncology*, 2007.

[34] Ilan Leibovitch, Ronan Lev, Yoram Mor, Jacob Golomb, Zohar A Dotan, and Jacob Ramon. Extensive Necrosis in Renal Cell Carcinoma Specimens: Potential Clinical and Prognostic Implications. *IMAJ*, 3:563–565, August 2001.

[35] Cédric Lemaréchal, Roger Fjørtoft, Philippe Marthon, and Eliane Cubero-Castan. Comments on "Geodesic Saliency of Watershed Contours and Hierarchical Segmentation". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(7), July 1998.

[36] Daw-Tung Lin, Chung-Chih Lei, and Siu-Wan Hung. Computer-Aided Kidney Segmentation on Abdominal CT Images. *IEEE Transactions on Information Technology in Biomedicine*, 10(1):59–65, January 2006.

[37] Steven Lobregt and Max A. Viergever. A Discrete Dynamic Contour Model. *IEEE Transactions on Medical Imaging*, 14(1), March 1995.

[38] William E Lorensen and Harvey E Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, volume 21, pages 163–169, New York, NY, USA, July 1987. ACM Press.

[39] Nadia Magnenat-Thalmann, Sunil Hadap, and Prem Kalra. State of the Art in Hair Simulation. In *Proceedings of the International Workshop on Human Modeling and Animation*, pages 3–9, 2000.

[40] Beatriz Marcotegui and Serge Beucher. Fast Implementation of Waterfall Based on Graphs. In *Mathematical Morphology: 40 Years On*. Springer Netherlands, 2005.

[41] A W McLean and S A Cameron. The virtual springs method: Path planning and collision avoidance for redundant manipulators. *Int. J. Robotic Research*, 15(4):300–319, August 1996.

[42] A Meijster and J Roerdink. A Disjoint Set Algorithm for the Watershed Transform. In *Proceedings of the 9th European Signal Processing Conference (EUSIPCO '98)*, 1998.

[43] M M Melicow. Classification of renal neoplasms: a clinical and pathological study based on 199 cases. *Journal of Urology*, 51:333–385, 1944.

[44] J V Miller. On GDM's: Geometrically deformed models for the extraction of closed shapes from volume data. Master's thesis, Rensselaer Polytechnic Institute, 1990.

[45] J V Miller, D E Breen, and M J Wozny. Extracting geometric models through constraint minimization. Technical Report 90024, Rensselaer Polytechnic Institute, 1990.

[46] Laurent Najman and Michel Schmitt. Geodesic Saliency of Watershed Contours and Hierarchical Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(12), December 1996.

[47] Victor Osma-Ruiz, Juan I Godino-Llorente, Nicolas Saenz-Lechon, and Pedro Gomez-Vilda. An improved watershed algorithm based on efficient computation of shortest paths. *Pattern Recognition*, 40(3):1078–1090, June 2006.

[48] Hyunjin Park, Peyton H Bland, and Charles R Meyer. Construction of an Abdominal Probabilistic Atlas and its Application in Segmentation. *IEEE Transactions on Medical Imaging*, 22(4):483–492, April 2003.

[49] Luis Patino. Fuzzy relations applied to minimize over segmentation in watershed algorithms. *Pattern Recognition*, 26:819–828, 2005.

[50] Pietro Perona and Jitendra Malik. Scale-Space and Edge Detection Using Anisotropic Diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[51] Dzung L Pham, Chenyang Xu, and Jerry L Prince. Current Methods in Medical Image Segmentation. *Annual Review of Biomedical Engineering*, 2:315–37, 2000.

[52] Stephen M Pizer, P Thomas Fletcher, Sarang Joshi, Andrew Thall, James Z Chen, Yonatan Fridman, Daniel S Fritsch, A Graham Gash, John M Glotzer, Michael R Jiroutek, Conglin Lu, Keith E Muller, Gregg Tracton, Paul Yushkevich, and Edward L Chaney. Deformable M-Reps for 3D Medical Image Segmentation. *International Journal of Computer Vision*, 55(2/3):85–106, 2003.

[53] R Pohle and K Toennies. Segmentation of Medical Images Using Adaptive Region Growing. In *Proceedings of SPIE (Medical Imaging)*, volume 4322, pages 1337–1346, 2001.

[54] C Rambabu and Indrajit Chakrabarti. An efficient immersion-based watershed transform method and its prototype architecture. *Journal of Systems Architecture*, 53:210–226, 2007.

[55] Eirik Roald Ree. Segmentation of Kidneys from MR-Images. Undergraduate thesis, Norwegian University of Science and Technology, 2005.

[56] T W Ridler and S Calvard. Picture Thresholding Using an Iterative Selection Method. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8(8), August 1978.

[57] Michel Schmitt. Response to the Comment on "Geodesic Saliency of Watershed Contours and Hierarchical Segmentation". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(7), July 1998.

[58] M Schouman, A Warter, M Roos, and C Bollack. Renal Cell Carcinoma: Statistical Study of Survival Based on Pathological Criteria. *World Journal of Urology*, 2:109–113, 1984.

[59] Shomik Sengupta, Christine M Lohse, Bradley C Leibovich, Igor Frank, R Houston Thompson, W Scott Webster, Horst Zincke, Michael L Blute, John C Cheville, and Eugene D Kwon. Histologic Coagulative Tumor Necrosis as a Prognostic Indicator of Renal Cell Carcinoma Aggressiveness. *Cancer*, 104(3):511–520, August 2005.

[60] Mehmet Sezgin and Bülent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, 13(1):146–165, January 2004.

[61] Anna Sokol. Modeling Hair Movement with Mass-Springs. Computer Science Department, SUNY Stony Brook.

[62] Luc Soler, Herve Delingette, Gregoire Malandain, Johan Montagnat, Nicholas Ayache, Christophe Koehl, Oliver Dourthe, Benoit Malassagne, Michelle Smith, Didier Mutter, and Jacques Marescaux. Fully Automatic Anatomical, Pathological, and Functional Segmentation from CT Scans for Hepatic Surgery. *Computer Aided Surgery*, 6:131–142, 2001.

[63] Stanislav L Stoev. RaFSi - A Fast Watershed Algorithm Based on Rainfalling Simulation. In *Proceedings of the 8th International Conference on Computer Graphics, Visualization and Interactive Digital Media (WSCG '00)*, 2000.

[64] Hasan Dogu Taskiran and Ugur Güdükbay. Physically-based simulation of hair strips in real-time. In *WSCG (Short Papers)*, pages 153–156, 2005.

[65] Wala Touhami, Djamal Boukerroui, and Jean-Pierre Cocquerez. Fully Automatic Kidneys Detection in 2D CT Images: A Statistical Approach. In *Proceedings of the 8th International Conference on Medical Image Computing and Computer-Assisted Intervention - Part I*, pages 262–269, 2005.

[66] Baigalmaa Tsagaan, Akinobu Shimizu, Hidefumi Kobatake, and Kunihisa Miyakawa. An Automated Segmentation Method of Kidney Using Statistical Information. In *Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention - Part I*, pages 556–563, 2002.

[67] Du-Yih Tsai and Nobutaka Tanahashi. Neural-Network-Based Boundary Detection of Liver Structure. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 6, pages 3484–3489, 1994.

[68] Lav R Varshney. Abdominal Organ Segmentation in CT Scan Images: A Survey. Cornell University, August 2002.

[69] Irina Voiculescu. Splines Sets V-VI. *Splines and Computational Geometry* course, Oxford University Computing Laboratory, 2005.

[70] Kelly Ward, Nico Galoppo, and Ming C. Lin. Modeling Hair Influenced by Water and Styling Products. In *CASA '04: Proceedings of Computer Animation and Social Agents*, pages 207–214, Geneva, Switzerland, 2004.

[71] Kelly Ward, Ming C. Lin, Joohi Lee, Susan Fisher, and Dean Macri. Modeling Hair Using Level-of-Detail Representations. In *CASA '03: Proceedings of the 16th International Conference on Computer Animation and Social Agents*, page 41, Washington DC, USA, 2003. IEEE Computer Society.

[72] S Wegner, T Harms, J H Builtjes, H Oswald, and E Fleck. The Watershed Transformation for Multiresolution Image Segmentation. In *Image Analysis and Processing*, pages 31–36. Springer Berlin/Heidelberg, 1995.

[73] Andrew K C Wong and P K Sahoo. A Gray-Level Threshold Selection Method Based on Maximum Entropy Principle. *IEEE Transactions on Systems, Man and Cybernetics*, 19(4), July/August 1989.

[74] James Worrell. Data Structures for Disjoint Sets. *Advanced Data Structures and Algorithms* course notes, Oxford University, 2006.

[75] Ziji Wu and John M. Sullivan Jr. Multiple material marching cubes algorithm. *International Journal for Numerical Methods in Engineering*, 58(2):189–207, July 2003.

[76] Chenyang Xu and Jerry L Prince. Snakes, Shapes and Gradient Vector Flow. *IEEE Transactions on Image Processing*, 7(3):359–369, March 1998.

[77] Zhan Xu and Xue Dong Yang. V-HairStudio: An Interactive Tool for Hair Design. *IEEE Computer Graphics and Applications*, 21(3):36–43, 2001.

[78] Terry S Yoo, editor. *Insight into Images*. A K Peters, 2004.

[79] Y J Zhang. A Survey on Evaluation Methods for Image Segmentation. *Pattern Recognition*, 29(8):1335–1346, 1996.